რომან სამხარაძე

Visual C++/CLI .NET

საქართველოს ტექნიკური უნივერსიტეტი

რომან სამხარაძე

V i s u a l C++/CLI.N E T

რეგისტრირებულია სტუ-ის
სარედაქციო-საგამომცემლო საბჭოს
მიერ. 29.07.2011, ოქმი №3

თბილისი
2012

სახელმძღვანელოში გადმოცემულია Microsoft Visual Studio .NET გარემოში პროგრამების შემუშავების საკითხები. დაწვრილებითაა განხილული C++ ენის საფუძვლები, მონაცემთა ტიპები, მმართველი ოპერატორები, სტრუქტურირებული ტიპები. გადმოცემულია ობიექტზე ორიენტირებული დაპროგრამების პრინციპები: ინკაფსულაცია, პოლიმორფიზმი და მემკვიდრეობითობა. განხილულია განსაკუთრებული სიტუაციები, ფაილებთან და კატალოგებთან მუშაობისა და CLI დაპროგრამების საკითხები. წინა გამოცემისაგან განსხვავებით წიგნში დამატებულია მონაცემთა ბაზებთან მუშაობის საკითხები. აღწერილია ADO.NET კლასები, ნაჩვენებია, თუ როგორ უნდა ვიმუშაოთ SQL სერვერზე მოთავსებულ მონაცემთა ბაზასთან C++ პროგრამიდან Active Data Objects ბიბლიოთეკის კლასების გამოყენებით .NET Framework პლატფორმისათვის. წიგნში უხვადაა მაგალითები და ამოცანები ამოხსნებით.

განკუთვნილია კომპიუტერული სისტემებისა და ქსელების სპეციალობის ბაკალავრებისა და მაგისტრებისათვის, აგრეთვე, დაპროგრამების შესწავლის ნებისმიერი მსურველისათვის.

რეცენზენტი ტექნიკის მეცნიერებათა კანდიდატი სრული პროფესორი მ. ანდღულაძე
ტექნიკის მეცნიერებათა კანდიდატი ასოცირებული პროფესორი ლ. გაჩეჩილაძე

ISBN 978-9941-20-002-1

http://gtu.ge/publishinghouse/

წიგნს ვუძღვნი

ჩემ შვილებს ანას და საბას

სარჩევი

წინასიტყვაობა

Visual Studio .NET წარმოადგენს დაპროგრამების თანამედროვე ტექნოლოგიას. მისი საშუალებით შესაძლებელია სხვადასხვა ტიპის პროგრამა-დანართების შემუშავება, როგორიცაა, Windows დანართები, ქსელური პროგრამები, მონაცემთა ბაზები და ა.შ. ის განკუთვნილია ქსელში ჩართულ კომპიუტერებთან და მოწყობილობებთან სამუშაოდ.

წიგნის მიზანია მკითხველს შეასწავლოს C++/CLI ენა, ობიექტზე ორიენტირებული დაპროგრამების საფუძვლები და განკუთვნილია დამწყები პროგრამისტებისათვის.

სახელმძღვანელო წარმოადგენს მეორე, გადამუშავებულ გამოცემას. მასში გადმოცემულია C++ ენის გაფართოება - C++/CLI, რომელიც რეალიზებულია .NET Framework გარემოში და აქვს მნიშვნელოვანი უპირატესობები C++ ენის წინა სტანდარტებთან შედარებით. წინა გამოცემისაგან განსხვავებით წიგნში დამატებულია მონაცემთა ბაზებთან მუშაობის საკითხები. აღწერილია ADO.NET კლასები, ნაჩვენებია, თუ როგორ უნდა ვიმუშაოთ SQL სერვერზე მოთავსებულ მონაცემთა ბაზასთან C++ პროგრამიდან Active Data Objects ბიბლიოთეკის კლასების გამოყენებით .NET Framework პლატფორმისათვის.

წიგნში აღწერილია Microsoft Visual Studio .NET 2010 სისტემა.

აუცილებელი პროგრამული უზრუნველყოფაა - Windows XP - SP/3, Windows 7, Visual Studio.NET 2008/2010.

წიგნი C++/CLI ენაზე დაპროგრამების საკითხთა ქართულ ენაზე გადმოცემის ერთ-ერთი პირველი მცდელობაა. ამიტომ, ის არ არის დაზღვეული ხარვეზებისაგან. ავტორი მადლიერებით მიიღებს წიგნში გადმოცემული მასალის დახვეწისა და სრულყოფის მიზნით გამოთქმულ შენიშვნებსა და მოსაზრებებს. ისინი შეგიძლიათ მომაწოდოთ მისამართებზე:
samkharadze.roman@gmail.com, rsamkharadze@mail.ru .

საქართველოს ტექნიკური უნივერსიტეტის
ინფორმატიკისა და მართვის სისტემების ფაკულტეტის
კომპიუტერული ინჟინერიის დეპარტამენტის
სრული პროფესორი, ტექნიკის მეცნიერებათა დოქტორი

რომან სამხარაძე

I ნაწილი. C++ ენა

თავი 1. შესავალი

დაპროგრამების ისტორიის მოკლე მიმოხილვა

დაპროგრამების ენები გამოიყენება ისეთი მრავალფეროვანი ამოცანების გადასაწყვეტად, როგორიცაა მონაცემთა საინფორმაციო სისტემის მართვა, რთული მათემატიკური და ეკონომიკური ამოცანის გადაწყვეტა, მედიცინა და ა.შ. C++ ენა, რომელიც შეიმუშავა კომპანია Microsoft-მა, მთლიანად პასუხობს დაპროგრამების თანამედროვე სტანდარტებს და განკუთვნილია Windows გარემოში მომუშავე თანამედროვე კომპიუტერული სისტემების შემუშავებისათვის.

დაპროგრამების ენებს შორის არსებობს კავშირი. ყოველი ახალი ენა ადრე შექმნილი ენებისაგან მემკვიდრეობით იღებს გარკვეულ თვისებებს. კერძოდ, C++ ენამ ბევრი სასარგებლო თვისება C ენისაგან მემკვიდრეობით მიიღო.

C ენა შეიქმნა ნიუ-ჯერსის შტატის ქალაქ მიურეი-ჰილის კომპანია Bell Laboratories-ის სისტემური პროგრამისტის დენის რიჩის მიერ 1972 წელს. თითქმის მთლიანად ამ ენაზე დაიწერა Unix, Windows და სხვა ოპერაციული სისტემების ბირთვი. შემდგომში, პროგრამების ზომისა და სირთულის ზრდამ საკმაოდ გააძნელა მათთან მუშაობა. გამოსავალი იყო სტრუქტურულ დაპროგრამებაზე გადასვლა. სწორედ C გახდა 1980 წლებიდან ყველაზე ხშირად გამოყენებადი სტრუქტურული დაპროგრამების ენა.

დაპროგრამების განვითარებასთან ერთად კვლავ დადგა დიდი ზომის პროგრამებთან მუშაობის პრობლემა. აუცილებელი გახდა ახალი მიდგომების შემუშავება. ერთ-ერთი მათგანია ობიექტზე ორიენტირებული დაპროგრამება (ოოდ). ის იძლევა დიდი ზომის პროგრამებთან ეფექტური მუშაობის შესაძლებლობას. შესაბამისად, 1974 წელს იმავე კომპანია Bell Laboratories-ში ბიარნ სტრაუსტრუპმა (Bjarne Stroustrup) შექმნა C ენის ობიექტზე ორიენტირებული ვერსია - "C კლასებით". მას 1983 წლიდან C++ დაერქვა. C++ მთლიანად მოიცავს C ენას და შეიცავს ობიექტზე ორიენტირებული დაპროგრამების შესაძლებლობებს. 1990 წლიდან იწყება C++ ენის მასობრივი გამოყენება და ის ხდება ყველაზე პოპულარული დაპროგრამების ენებს შორის.

ამ ენამ პირველი ცვლილება 1985 წელს განიცადა, მეორე კი - 1990 წელს. 1994 წელს მესამე ცვლილების შედეგად მიღებულ იქნა C++ ენის ერთიანი საერთაშორისო სტანდარტი. შემდეგ იგი მნიშვნელოვნად გაფართოვდა. მას დაემატა ალექსანდრე სტეპანოვის მიერ შემუშავებული სტანდარტული შაბლონების ბიბლიოთეკა (Standart Template Library, STL).

წიგნში განხილულია C++ ენის ახალი ვერსია - C++/CLI (Common Language Infrastructure, საერთო ენის ინფრასტრუქტურა), რომელიც დამტკიცებულია ECMA (European Association for Standartizing Information and Computer Systems, საინფორმაციო და გამოთვლითი სისტემების სტანდარტიზების ევროპის ასოციაცია) სტანდარტით.

.NET Framework გარემო

.NET Framework გარემო არის Visual C++ სისტემის ცენტრალური ნაწილი და ორი ელემენტისგან შედგება: ***საერთოენობრივი შესრულების გარემო*** **(Common Language Runtime, CLR)**, რომელშიც ჩვენი პროგრამა სრულდება, და ბიბლიოთეკების ნაკრები, რომელსაც .NET

Framework ***კლასების ბიბლიოთეკა*** ეწოდება. ბიბლიოთეკების ეს ნაკრები საჭიროა .NET პროგრამების CLR გარემოში შესრულებისას. .NET პროგრამა შეიძლება დაწერილი იყოს C++, C# ან Visual Basic ენებზე და ისინი ერთსა და იმავე .NET ბიბლიოთეკებს იყენებენ.

.NET Framework გარემო:

- აადვილებს პროგრამისა და Web-სამსახურის აგებას;
- იძლევა მნიშვნელოვან უპირატესობას კოდის საიმედოობისა და უსაფრთხოების კუთხით;
- იძლევა C++ ენაზე დაწერილი პროგრამის ინტეგრირების შესაძლებლობას დაპროგრამების იმ ენებთან, რომლებიც .NET Framework გარემოზეა ორიენტირებული.

საერთოენობრივი შესრულების გარემო

CLR არის იმ პროგრამების საერთოენობრივი შესრულების გარემო, რომლებიც დაწერილია **საერთო ენის ინფრასტრუქტურის (Common Language Infrastructure, CLI)** გამოყენებით. CLR-ი არის CLI სპეციფიკაციის რეალიზება, რომელიც Microsoft Windows-ის მართვის ქვეშ ფუნქციონირებს.

ფაილს, რომელიც შეიცავს C++ პროგრამის საწყის კოდს, აქვს .cpp, სათაურის ფაილის გაფართოება კი არის .h. ამ ფაილის კომპილაციას მანქანურ კოდებად ასრულებს პროგრამა, რომელსაც ***კომპილატორი*** ეწოდება. C++ პროგრამების კომპილაციის შედეგად მიიღება ფაილი, რომელიც შეიცავს ***შუალედურ ენას*** (Microsoft Intermediate Language, MSIL). ის შედგება გადასატანი ინსტრუქციების ნაკრებისაგან, რომელიც არაა დამოკიდებული კონკრეტული პროცესორის ინსტრუქციების ნაკრებზე. პროგრამის გაშვებისას CLR სისტემა შუალედურ კოდს გარდაქმნის შესრულებად კოდად. MSIL ენაში კომპილირებული პროგრამა შეიძლება ნებისმიერ ოპერაციულ სისტემაში შესრულდეს, თუ არის დაყენებული შესაბამისი ვერსიის Net FrameWork გარემო.

MSIL ენას შესრულებად კოდში გარდაქმნის JIT ***ოპერატიული კომპილატორი*** (just in time). C++ პროგრამის გაშვებისას CLR სისტემა ააქტიურებს JIT კომპილატორს, რომელიც MSIL ენას გარდაქმნის მოცემული პროცესორის შიგა კოდად. ამასთან, პროგრამის ნაწილების გარდაქმნა საჭიროების მიხედვით სრულდება.

CLI აგრეთვე, შეიცავს მონაცემთა ტიპების საერთო ნაკრებს, რომელსაც ***ტიპების საერთო სისტემა*** **(Common Type System, CTS)** ეწოდება. CTS-ს იყენებენ ნებისმიერ ენაზე დაწერილ პროგრამაში, რომელიც CLR გარემოში სრულდება. CTS-ი შეიცავს წინასწარ განსაზღვრული ტიპების ნაკრებს. შეგვიძლია, აგრეთვე, განვსაზღვროთ მონაცემთა საკუთარი ტიპებიც, მაგრამ ისინი CLR-თან შეთანხმებული უნდა იყოს. მონაცემების წარმოდგენისათვის ტიპების სტანდარტიზებული სისტემა სხვადასხვა ენაზე დაწერილ პროგრამებს საშუალებას აძლევს მონაცემები უნიფიცირებული საშუალებით დაამუშაონ და უზრუნველყოფს სხვადასხვა ენაზე დაწერილი პროგრამების ინტეგრირების შესაძლებლობას ერთ პროგრამად.

პროგრამების უსაფრთხოება და საიმედოობა CLR-ში მნიშვნელოვნადაა გაზრდილი იმის ხარჯზე, რომ მეხსიერების დინამიკური გამოყოფა და გათავისუფლება მთლიანად ავტომატიზებულია.

.NET Framework გარემოში მუშაობისას ჩვენ ვქმნით ***მართვად კოდს*** (managed code), რომელსაც მართავს CLR სისტემა და რომელიც ამავე სისტემაში მუშაობს. მართვად კოდს შემდეგი უპირატესობები აქვს: მეხსიერების მართვა, სხვადასხვა ენის შეთავსების შესაძლებლობა, მონაცემების გადაცემის უსაფრთხოების მაღალი დონე, ვერსიის კონტროლის უზრუნველყოფა და პროგრამული უზრუნველყოფის კომპონენტების ადვილად ურთიერთქმედების საშუალება.

არსებობს, აგრეთვე, ***არამართვადი კოდი*** (unmanaged code), რომელსაც CLR სისტემა არ

ასრულებს. .NET Framework გარემოს შექმნამდე ყველა კოდი არამართვადი იყო. ამჟამად, ორივე სახის კოდს შეუძლია ერთად მუშაობა. მაგალითად, C++ ქმნის მართვად კოდს, რომელსაც შეუძლია არამართვად კოდთან ურთიერთქმედება.

თუ ჩვენ მიერ შექმნილ კოდს გამოიყენებენ სხვა ენებზე დაწერილი პროგრამები, მაშინ მათი მაქსიმალური თავსებადობისათვის უნდა დავიცვათ ***საერთოენობრივი სპეციფიკაცია*** (Common Language Specification, CLS). იგი აღწერს სხვადასხვა ენის საერთო მახასიათებლებს.

C++ კოდს, რომელიც CLR-ის მართვით სრულდება, ***მართვადი*** **C++** ეწოდება. CLR-პროგრამებში მეხსიერების გათავისუფლება, რომელიც დინამიკურად გამოიყოფა მონაცემების მოსათავსებლად, ავტომატურად ხორციელდება, რაც გამორიცხავს „მშობლიური“ პროგრამების შეცდომების ძირითად წყაროს. C++ კოდს, რომელიც სრულდება CLR გარემოს გარეთ ***არამართვადი*** **C++** ეწოდება, რადგან CLR გარემო მის შესრულებაში არ მონაწილეობს. არამართვად C++-ში ჩვენ თვითონ უნდა ვიზრუნოთ მეხსიერების გამოყოფასა და გათავისუფლებაზე პროგრამის შესრულებისას, აგრეთვე უსაფრთხოებაზე. არამართვად C++-ს სხვანაირად „მშობლიური C++“ ეწოდება, რადგან ის კომპილირდება უშუალოდ მშობლიურ მანქანურ კოდში.

პროგრამა, აგრეთვე, შეიძლება შედგებოდეს ***ნაწილობრივ მართვადი კოდისა*** და ***ნაწილობრივ მშობლიური კოდისგან***. ასეთი პროგრამა უმჯობესია შევადგინოთ მხოლოდ მაშინ, როდესაც არსებული მშობლიური C++ პროგრამა გვინდა ვამუშაოთ CLR-ის მართვით.

როგორც აღვნიშნეთ, არსებობს Windows-პროგრამების ორი ძირითადი ვარიანტი: პროგრამა, რომელიც სრულდება CLR-ის მართვით და პროგრამა, რომელიც უშუალოდ კომპილირდება ოპერაციული სისტემის „მშობლიურ“ მანქანურ კოდად. Windows-პროგრამებისთვის, რომლებიც CLR-ზეა ორიენტირებული, მომხმარებლის გრაფიკული ინტერფეისის (Graphical User Interface, GUI) აგების ბაზად გამოიყენება Windows Forms გარსი, რომელსაც წარმოგვიდგენს .NET Framework საბაზო კლასების ბიბლიოთეკა. Windows Forms გარსის გამოყენება უზრუნველყოფს GUI-ის სწრაფ შემუშავებას, რადგან ჩვენ მას გრაფიკულად ვირჩევთ სტანდარტული კომპონენტებიდან და ვიღებთ მის ავტომატურად გენერირებულ კოდს. C++ ენის „მშობლიური“ შესრულებადი კოდის მიღების ერთ-ერთი გზაა Microsoft Foundation Classes (MFC) კლასების ბიბლიოთეკის გამოყენება.

CLR-ს მნიშვნელოვანი უპირატესობები აქვს C++ ენის „მშობლიურ“ გარემოსთან შედარებით. CLR-ში მომუშავე პროგრამისთვის უზრუნველვყოფილია მაღალი უსაფრთხოების დონე და შემცირებულია პოტენციური შეცდომების დაშვების ალბათობა, რომლებიც შესაძლებელია ISO/ANSI C++ სტანდარტის გამოყენებისას. გარდა ამისა, CLR-ი გამორიცხავს შეუთავსებლობას დაპროგრამების მაღალი დონის ენებს შორის მიზნობრივი გარემოს სტანდარტიზაციის ხარჯზე, რომლისთვისაც სრულდება კომპილირება. ეს საშუალებას გვაძლევს შევასრულოთ C++ ენაზე დაწერილი მოდულის (პროგრამის) კომბინირება სხვა ენებზე, მაგალითად C#-ზე, დაწერილ მოდულთან.

C++ ენის სტანდარტები

Visual Studio უზრუნველყოფს C++ ენის ორი სხვადასხვა, მაგრამ ერთმანეთთან მჭიდროდ დაკავშირებული ვერსიის მუშაობას. ეს ვერსიებია:

- **ISO/ANSI C++** (International Standards Organization, სტანდარტების საერთაშორისო ორგანიზაცია/ American National Standards Institute, სტანდარტების ამერიკის ეროვნული ინსტიტუტი). ეს არის C++ ენის თავდაპირველი სტანდარტი. ის აღწერს C++ ენის იმ ვერსიას, რომელიც არსებობს 1998 წლიდან და უზრუნველყოფილია ოპერაციული სისტემებისა და კომპიუტერული პლატფორმების უმეტესობით. ამ სტანდარტით

განსაზღვრულ C++ ენაზე დაწერილ პროგრამას **C++-*ის მშობლიური პროგრამა*** ეწოდება. ISO/ANSI C++ ენაზე დაწერილი პროგრამის გადატანა ერთი პლატფორმიდან მეორეზე შედარებით ადვილია. გადატანის სირთულეს განსაზღვრავს პროგრამაში გამოყენებული ბიბლიოთეკური ფუნქციები;

- **C++/CLI** არის C++ ენის ვერსია, რომელიც აფართოებს ISO/ANSI სტანდარტს C++ ენის საერთო ინფრასტრუქტურის (CLI) უკეთესი უზრუნველყოფის მიზნით. ამ სტანდარტის პირველი ვარიანტი გამოვიდა 2003 წელს და შემუშავებული იყო Microsoft-ის მიერ C++ პროგრამების შესასრულებლად .NET Framework გარემოს მართვით. პროგრამა, რომელიც სრულდება CLR-ის მართვით და რომელიც რეალიზებულია C++ ენის C++/CLI გაფართოებული ვერსიის საშუალებით **CLR *პროგრამა*** ან **C++/CLI *პროგრამა*** ეწოდება.

ISO/ANSI C++ სტანდარტი განკუთვნილია მაღალმწარმოებლური პროგრამების (application, приложение, პროგრამა-დანართი) შესამუშავებლად. ისინი სრულდება როგორც მშობლიური პროგრამები (არამართვადი C++). C++/CLI ვერსია სპეციალურად შემუშავებულია .NET Framework გარემოსთვის და Windows Forms პროგრამების (მართვადი C++) შემუშავების მძლავრი ინსტრუმენტია. დაპროგრამების სისწრაფე და სიმარტივე C++/CLI ვერსიას უფრო მიმზიდველს ხდის როგორც დამწყები, ისე პროფესიონალი პროგრამისტებისთვის.

C++/CLI ენაზე დაწერილი პროგრამა საშუალებას გვაძლევს ვისარგებლოთ .NET Framework გარემოს უპირატესობებით, რომელიც ხშირად მიუწვდომელია ISO/ANSI C++ ენაზე დაწერილი პროგრამებისთვის. მართალია, C++/CLI არის ISO/ANSI C++ ვერსიის გაფართოება, მაგრამ იმისათვის, რომ პროგრამა მთლიანად შესრულდეს CLR-ის მართვით, მან უნდა დააკმაყოფილოს CLR-ის მოთხოვნები. ეს იმას ნიშნავს, რომ ISO/ANSI C++ ვერსიის ზოგიერთი საშუალების გამოყენება CLR-პროგრამაში არ შეიძლება. ასეთი შეზღუდვის ერთ-ერთი მაგალითია მეხსიერების გამოყოფისა და გათავისუფლების ISO/ANSI C++ ვერსიის საშუალებების აკრძალვა. ისინი შეუთავსებელია C++/CLI ვერსიასთან და მათ ნაცვლად უნდა გამოვიყენოთ მეხსიერების მართვის CLR-ის მექანიზმი. ეს კი, იმას ნიშნავს, რომ ჩვენ უნდა ვიმუშაოთ C++/CLI ვერსიის კლასებთან და არა C++ ენის მშობლიურ კლასებთან.

პროგრამის შემუშავების ინტეგრირებული გარემო

პროგრამის შემუშავების ინტეგრირებული გარემო (Integrated Development Environment, IDE), რომელიც თან ახლავს Visual C++ სისტემას, არის C++ პროგრამების შექმნის, კომპილირების, აწყობისა და ტესტირებისთვის განკუთვნილი გარემო. მისი კომპონენტებია: რედაქტორი, კომპილატორი, ამწყობი და ბიბლიოთეკები.

რედაქტორი (editor) არის ინტეგრირებული გარემო, რომელშიც შეგვიძლია C++ ენის საწყისი კოდის შექმნა და რედაქტირება. რედაქტორი ავტომატურად აღიქვამს C++ ენის საკვანძო სიტყვებს, შეაფერადებს მათ და აქვს ტექსტური რედაქტორისთვის დამახასიათებელი ფუნქციები.

კომპილატორი (compiler) საწყის კოდს გარდაქმნის ობიექტურ კოდად და გვაუწყებს კომპილაციისას აღმოჩენილი შეცდომების შესახებ. გამოსასვლელ ობიექტურ კოდს კომპილატორი ათავსებს ***ობიექტურ ფაილში***. ასეთი ფაილის გაფართოებაა .obj .

ამწყობი (builder) ახდენს კომპილატორით გენერირებული სხვადასხვა მოდულის კომბინირებას, უმატებს მათ საჭირო მოდულებს ბიბლიოთეკებიდან და აერთიანებს ერთ ფაილში. ამწყობს შეუძლია, აგრეთვე, შეცდომების აღმოჩენა და მათ შესახებ შეტყობინების გაცემა.

ბიბლიოთეკა (library) არის წინასწარ დაწერილი პროცედურების (პროგრამების) კოლექცია, რომლებიც აფართოებენ C++ ენის შესაძლებლობებს. ისინი შეგვიძლია ჩავრთოთ ჩვენ

პროგრამებში სტანდარტული და ხშირად გამოყენებადი ოპერაციების შესასრულებლად.

ობიექტზე ორიენტირებული დაპროგრამება

ჩვენ გარშემო ბევრი ობიექტია. მაგალითად, ავტომობილი, თვითმფრინავი, მატარებელი, მაღაზია, ავადმყოფი, ექიმი, სტუდენტი, გეომეტრიული ფიგურა და ა.შ. მსგავსი ობიექტები შეგვიძლია დავაჯგუფოთ კლასებად. მაგალითად, სხვადასხვა ავტომობილს ბევრი საერთო თვისება აქვს, ამიტომ ისინი შეგვიძლია ავტომობილის საერთო კლასში გავაერთიანოთ. ასევე სხვადასხვა ტიპის თვითმფრინავს ბევრი საერთო თვისება აქვს, ამიტომ ისინი შეგვიძლია თვითმფრინავის საერთო კლასში გავაერთიანოთ და ა.შ.

ობიექტზე ორიენტირებულ ენაში პროგრამები ორგანიზებულია მონაცემების გარშემო, ანუ ჩვენ განვსაზღვრავთ მონაცემებს და იმ პროგრამებს, რომლებიც ამ მონაცემებთან მუშაობენ. C++ ენა ეფუძნება ობიექტზე ორიენტირებული დაპროგრამების სამ პრინციპს პრინციპს: ინკაფსულაციის, პოლომორფიზმისა და მემკვიდრეობითობის. მოკლედ განვიხილოთ თითოეული მათგანი.

ინკაფსულაცია

ინკაფსულაცია (encapsulation) არის დაპროგრამების მექანიზმი, რომელიც აერთიანებს პროგრამის კოდსა და იმ მონაცემებს, რომლებთანაც ეს კოდი მუშაობს, აგრეთვე, გამიჯნავს მათ (კოდსა და მონაცემებს) სხვა პროგრამის მხრიდან მიმართვისაგან. ამით ხდება მათი დაცვა არასწორი გამოყენებისაგან. ობიექტზე ორიენტირებულ ენაში პროგრამის კოდი და მონაცემები ერთმანეთს ისე უკავშირდება, რომ ქმნიან ერთ ავტონომიურ სტრუქტურას, რომელსაც ობიექტი ეწოდება.

ობიექტში პროგრამის კოდი და მონაცემები სხვა ობიექტისათვის შეიძლება იყოს დახურული (private) ან ღია (public). დახურულ (პრივატულ) კოდთან და მონაცემებთან მიმართვა შეუძლია მხოლოდ ამავე ობიექტში აღწერილ კოდებს. ეს იმას ნიშნავს, რომ დახურულ კოდსა და მონაცემებს ვერ მივმართავთ პროგრამის სხვა ნაწილიდან, რომელიც ობიექტის გარეთაა მოთავსებული. ღია (საერთო წვდომის) კოდთან და მონაცემებთან მიმრთვა შესაძლებელია პროგრამის ნებისმიერი ნაწილიდან.

კლასი არის სტრუქტურა, რომელიც იყენებს ინკაფსულაციის პრინციპს და განსაზღვრავს კოდს და იმ მონაცემებს, რომლებთანაც ის მუშაობს. კლასი გამოიყენება ობიექტების აღსაწერად და შესაქმნელად. ობიექტები კლასის ეგზემპლარებია. კლასის შემადგენელ კოდსა და მონაცემებს კლასის წევრები ეწოდება, კლასის მიერ განსაზღვრულ მონაცემებს კი - ეგზემპლარის ცვლადები. კლასში განსაზღვრულ პროგრამის კოდს ფუნქცია ეწოდება.

პოლიმორფიზმი

პოლიმორფიზმი (polymorphism) არის დაპროგრამების მექანიზმი, რომელიც მსგავს ობიექტებს საშუალებას აძლევს ერთი ინტერფეისის გამოყენებით მიმართონ სხვადასხვა ფუნქცია. სხვა სიტყვებით რომ ვთქვათ, ერთი და იგივე სახელი შეიძლება რამდენიმე ფუნქციას ჰქონდეს, რომლებიც ერთსა და იმავე მოქმედებებს სხვადასხვა ტიპის მონაცემებზე ასრულებენ. ამ შემთხვევაში, არგუმენტის ტიპის მიხედვით სრულდება შესაბამისი ფუნქციის გამოძახება. კონკრეტულ ფუნქციას არგუმენტის ტიპის მიხედვით ირჩევს კომპილატორი. ამრიგად, პოლიმორფიზმი საშუალებას გვაძლევს რამდენიმე სახელის ნაცვლად გამოვიყენოთ ერთი. პოლიმორფიზმის უზრუნველყოფა ხდება ფუნქციების გადატვირთვის საშუალებით.

მემკვიდრეობითობა

მემკვიდრეობითობა (inheritance) არის დაპროგრამების მექანიზმი, რომლის საშუალებითაც ერთ ობიექტს მემკვიდრეობით გადაეცემა მეორე ობიექტის ელემენტები. ამავე დროს დაცულია იერარქიული სტრუქტურა მიმართული ზევიდან ქვევით. მაგალითად, კლასი **ლომი** არის **კატისებრნი** კლასის ნაწილი, რომელიც თავის მხრივ ეკუთვნის **მტაცებლები** კლასს. ეს უკანასკნელი კი არის **ცხოველები** კლასის ნაწილი. **ცხოველები** კლასს აქვს ისეთი მახასიათებლები, როგორიცაა ტყეში ცხოვრება და სხვა. იგივე მახასიათებლები შეგვიძლია გამოვიყენოთ **მტაცებლები** კლასის მიმართ, რომელსაც დამატებით აქვს თავისი სპეციფიკური მახასიათებლები, როგორიცაა ნადირობის უნარი და ა.შ., რომელიც მას სხვა ცხოველებისაგან განასხვავებს. აღნიშნული მახასიათებლების მატარებელია **კატისებრნი** კლასი. მას დამატებით აქვს ისეთი მახასიათებლები, რომლებიც მას სხვა მტაცებლებისაგან განასხვავებს. დაბოლოს, **ლომი** კლასი ყველა ზემოთ აღნიშნული მახასიათებლების მატარებელია და დამატებით შეიცავს მხოლოდ მისთვის დამახასიათებელ სპეციფიკურ მახასიათებლებს.

მემკვიდრეობითობის გამოყენება ობიექტს საშუალებას აძლევს განსაზღვროს ის ელემენტები, რომლებიც მხოლოდ მისთვისაა დამახასიათებელი. მემკვიდრე კლასი მშობელი (წინაპარი) კლასისაგან მემკვიდრეობით იღებს საერთო ელემენტებს. ამიტომ, მემკვიდრე კლასის გამოცხადებისას აღარ ხდება საჭირო მისი ყველა ელემენტის აღწერა.

თავი 2. C++ ენის საფუძვლები

პირველი პროგრამა

C++ ენაზე შევადგინოთ უმარტივესი პროგრამა, რომელიც შეასრულებს ორი მთელი რიცხვის შეკრებას.

```
{
//      პროგრამა 2.1
//      ეს არის ჩვენი პირველი პროგრამა
int ricxvi1, ricxvi2, jami;

ricxvi1 = 2;
ricxvi2 = 3;
jami    = ricxvi1 + ricxvi2;
}
```

როგორც ვხედავთ C++ ენაზე შედგენილი პროგრამა იწყება გამხსნელი ფიგურული ფრჩხილით "{" და მთავრდება დამხურავი ფიგურული ფრჩხილით "}". პროგრამის მეორე და მესამე სტრიქონებში მოთავსებულია კომენტარები, რომელებიც // სიმბოლოებით იწყება (კომენტარებს მოგვიანებით განვიხილავთ). მომდევნო სტრიქონში ხდება ცვლადების გამოცხადება. int სიტყვა (integer - მთელი) იწყებს ცვლადების გამოცხადებას და მიუთითებს, რომ ისინი მთელი რიცხვებია. მას მოსდევს ცვლადების სახელები ანუ იდენტიფიკატორები, რომლებიც ერთმანეთისაგან მძიმეებით გამოიყოფა. წერტილ-მძიმე ამთავრებს ცვლადების გამოცხადებას, აგრეთვე, ერთმანეთისაგან გამოყოფს ოპერატორებსა და ფუნქციებს (ფუნქცია არის მცირე ზომის პროგრამა, რომელიც გარკვეულ მოქმედებებს ასრულებს). ცვლადების სახელები აუცილებლად უნდა იწყებოდეს სიმბოლოთი და უმჯობესია შევარჩიოთ შინაარსიდან გამომდინარე. C++ ენაში ნებისმიერი ცვლადი გამოცხადებული უნდა იყოს მის გამოყენებამდე.

ცვლადების გამოცხადების შემდეგ ricxvi1 ცვლადს ენიჭება მნიშვნელობა - 2. "=" სიმბოლო არის მინიჭების ოპერატორი. ის მის მარჯვნივ მოთავსებულ მნიშვნელობას გადაწერს მის მარცხნივ მოთავსებულ ცვლადში. იგივე ეხება პროგრამის მომდევნო სტრიქონს. უკანასკნელ სტრიქონში jami ცვლადს ენიჭება ricxvi1 და ricxvi2 ცვლადების მნიშვნელობების ჯამი.

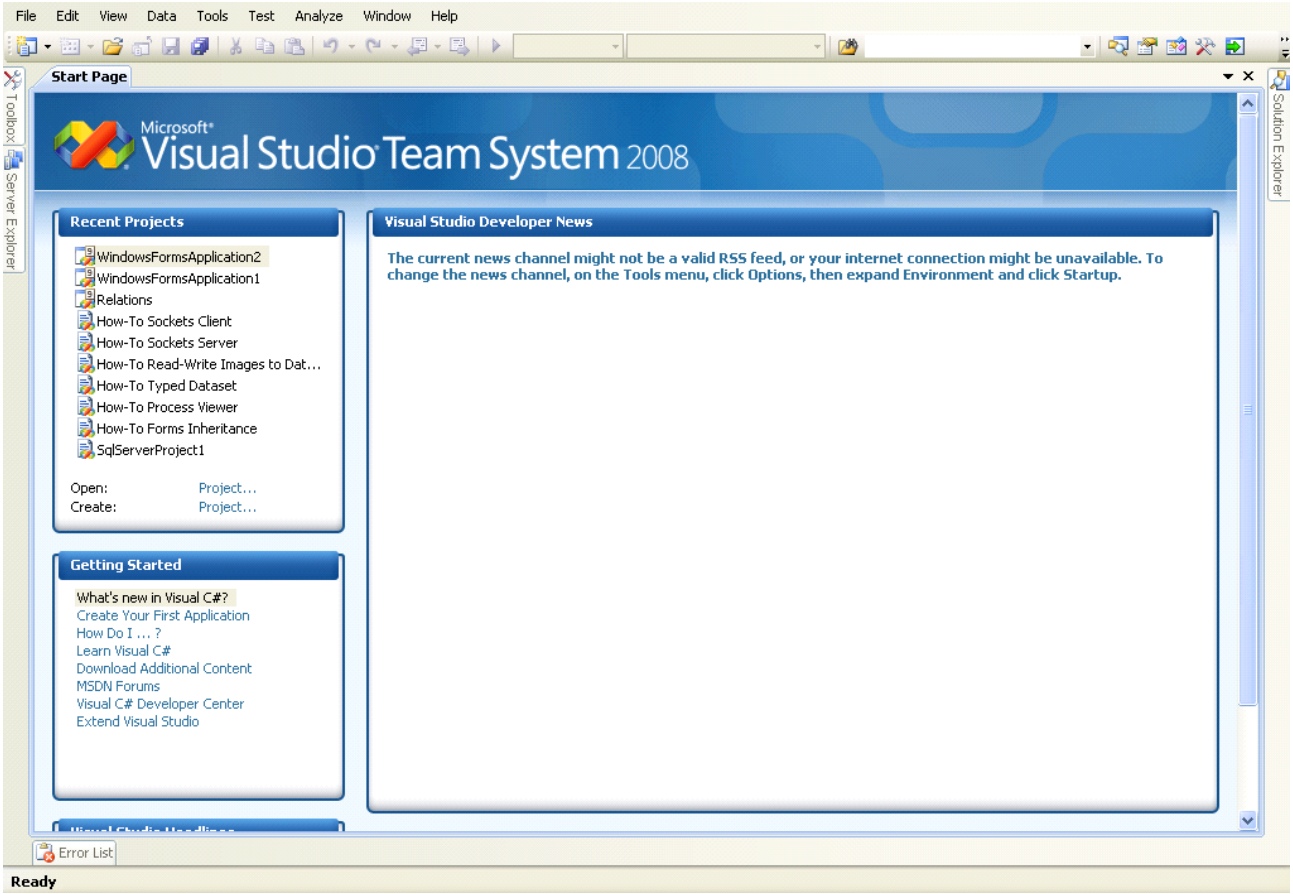
თვალსაჩინოების მიზნით პროგრამის თითოეულ სტრიქონში მოთავსებულია თითო ოპერატორი. თუმცა ერთ სტრიქონში შეიძლება რამდენიმე ოპერატორის მოთავსება. მთავარია, რომ ისინი ერთმანეთისაგან წერტილ-მძიმეებით იყოს გამოყოფილი.

ერთ-ერთი ნაკლი, რომელიც ამ პროგრამას აქვს ის არის, რომ სხვადასხვა რიცხვების შესაკრებად მოგვიწევს პროგრამის საწყისი კოდის შეცვლა, რაც არასწორია. გარდა ამისა, პროგრამას შედეგი ეკრანზე არ გამოაქვს. ამ ნაკლის აღმოსაფხვრელად გამოვიყენებთ ვიზუალურ კომპონენტებს, კერძოდ, მონაცემების შესატანად გამოვიყენებთ textBox კომპონენტს, გამოსატანად - label კომპონენტს, პროგრამის (ფუნქციის) შესასრულებლად კი - button კომპონენტს. მათი გამოყენებით 2.1 პროგრამა შემდეგ სახეს მიიღებს:

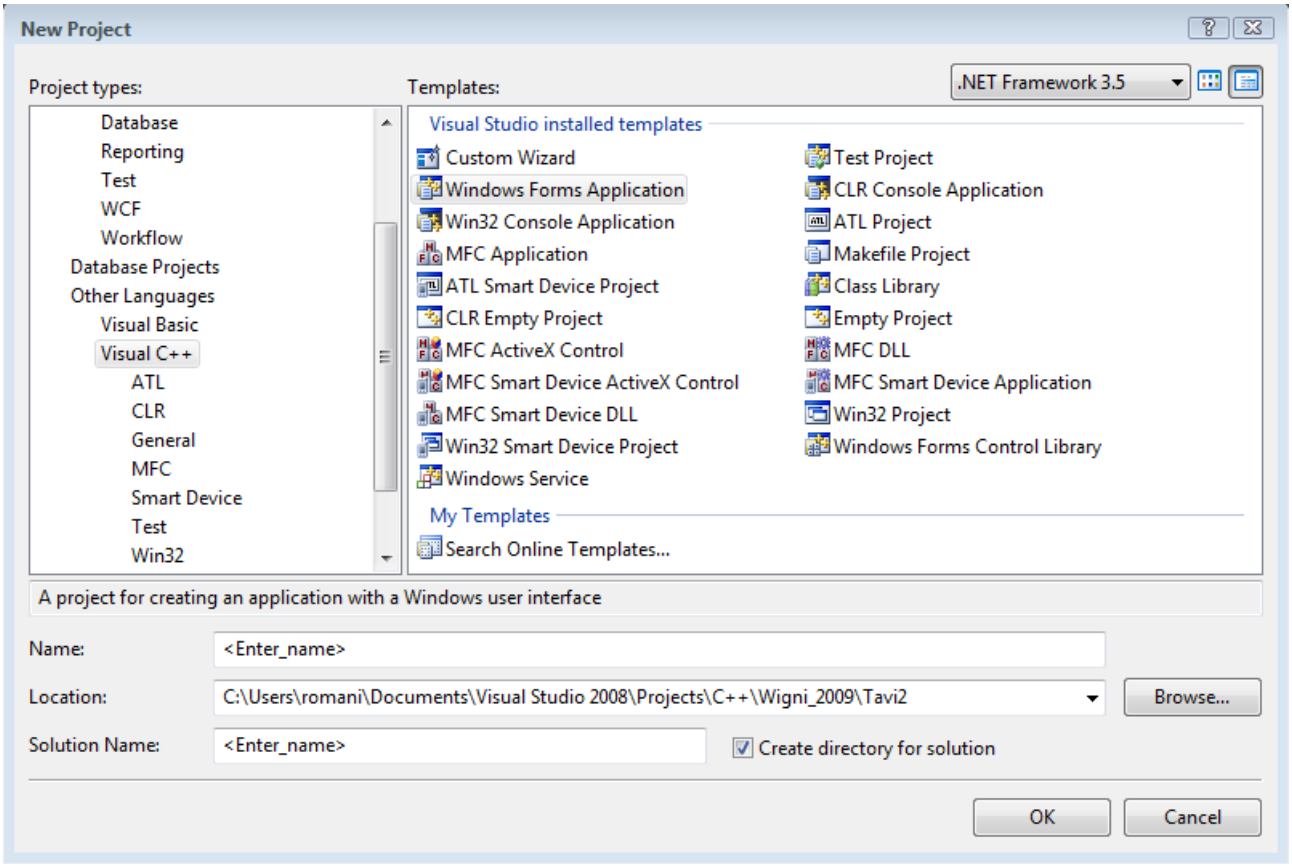
```
{
//      პროგრამა 2.2
//      ორი რიცხვის შეკრების პროგრამა
int ricxvi1, ricxvi2, jami;
```

```
ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
jami    = ricxvi1 + ricxvi2;
label5->Text = jami.ToString();
}
```

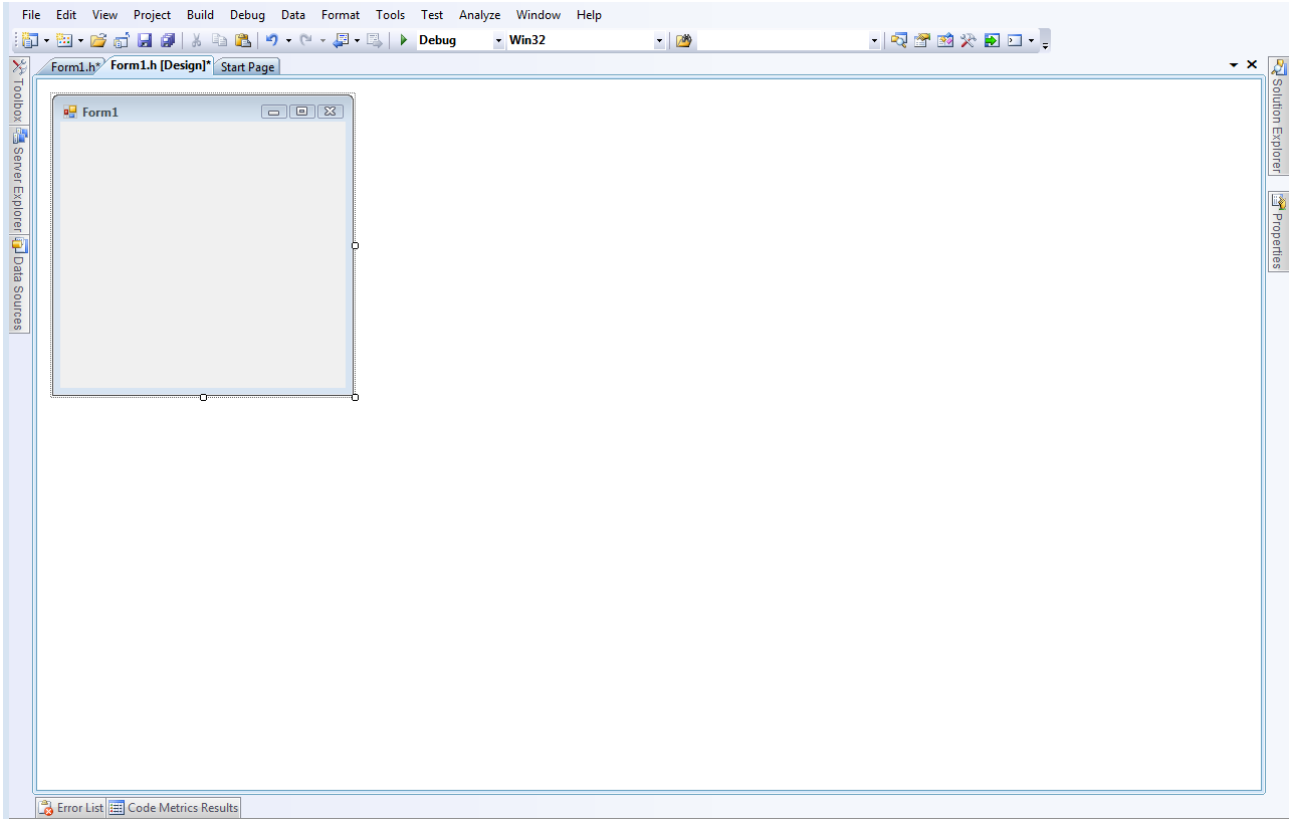
როგორც პროგრამიდან ჩანს, ricxvi1 ცვლადს ენიჭება textBox1 კომპონენტში შეტანილი მნიშვნელობა (მაგალითად, 5), რომელიც Convert::ToInt32 ფუნქციის საშუალებით მთელ რიცხვად გარდაიქმნება. საქმე ის არის, რომ textBox1 კომპონენტში შეტანილი მონაცემი მიენიჭება ამავე კომპონენტის Text თვისებას, რომელსაც სტრიქონული ტიპი აქვს. ამიტომ, ამ კომპონენტში შეტანილ ნებისმიერ მონაცემს სტრიქონული ტიპი ექნება. Convert::ToInt32 ფუნქცია თავის არგუმენტს, ჩვენ შემთხვევაში ესაა textBox1->Text, გარდაქმნის რიცხვად. შედეგად, textBox1 კომპონენტში შეტანილი მონაცემი, რომელიც სინამდვილეში სტრიქონია, მთელ რიცხვად გარდაიქმნება. ასეთი გარდაქმნა საჭიროა იმიტომ, რომ ricxvi1 არის მთელი ტიპის მქონე ცვლადი, ამიტომ მას უნდა მიენიჭოს მხოლოდ მთელი რიცხვი. იგივე ეხება მომდევნო სტრიქონებს. ზოგად შემთხვევაში კი მინიჭების ოპერატორის მარჯვნივ მოთავსებული გამოსახულების ტიპი დაყვანილი უნდა იყოს მინიჭების ოპერატორის მარცხნივ მოთავსებული ცვლადის ტიპზე.



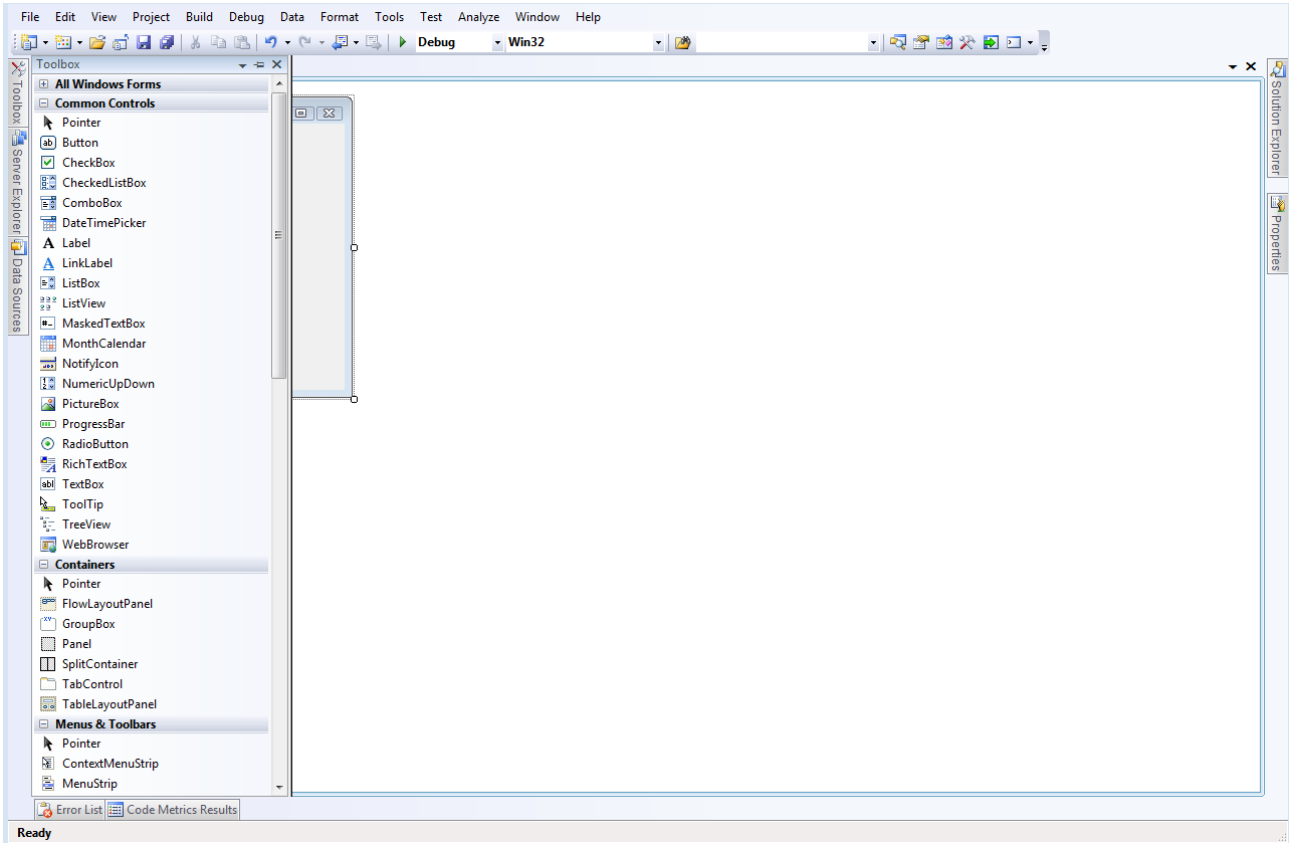
ნახ. 2.1.


New Project
Project types:
Templates:
.NET Framework 3.5
Database
Reporting
Test
WCF
Workflow
Database Projects
Other Languages
Visual Basic
Visual C++
ATL
CLR
General
MFC
Smart Device
Test
Win32
Visual Studio installed templates
Custom Wizard
Windows Forms Application
Win32 Console Application
MFC Application
ATL Smart Device Project
CLR Empty Project
MFC ActiveX Control
MFC Smart Device ActiveX Control
MFC Smart Device DLL
Win32 Smart Device Project
Windows Service
Test Project
CLR Console Application
ATL Project
Makefile Project
Class Library
Empty Project
MFC DLL
MFC Smart Device Application
Win32 Project
Windows Forms Control Library
My Templates
Search Online Templates...
A project for creating an application with a Windows user interface
Name:
<Enter_name>
Location:
C:\Users\romani\Documents\Visual Studio 2008\Projects\C++\Wigni_2009\Tavi2
Browse...
Solution Name:
<Enter_name>
Create directory for solution
OK
Cancel

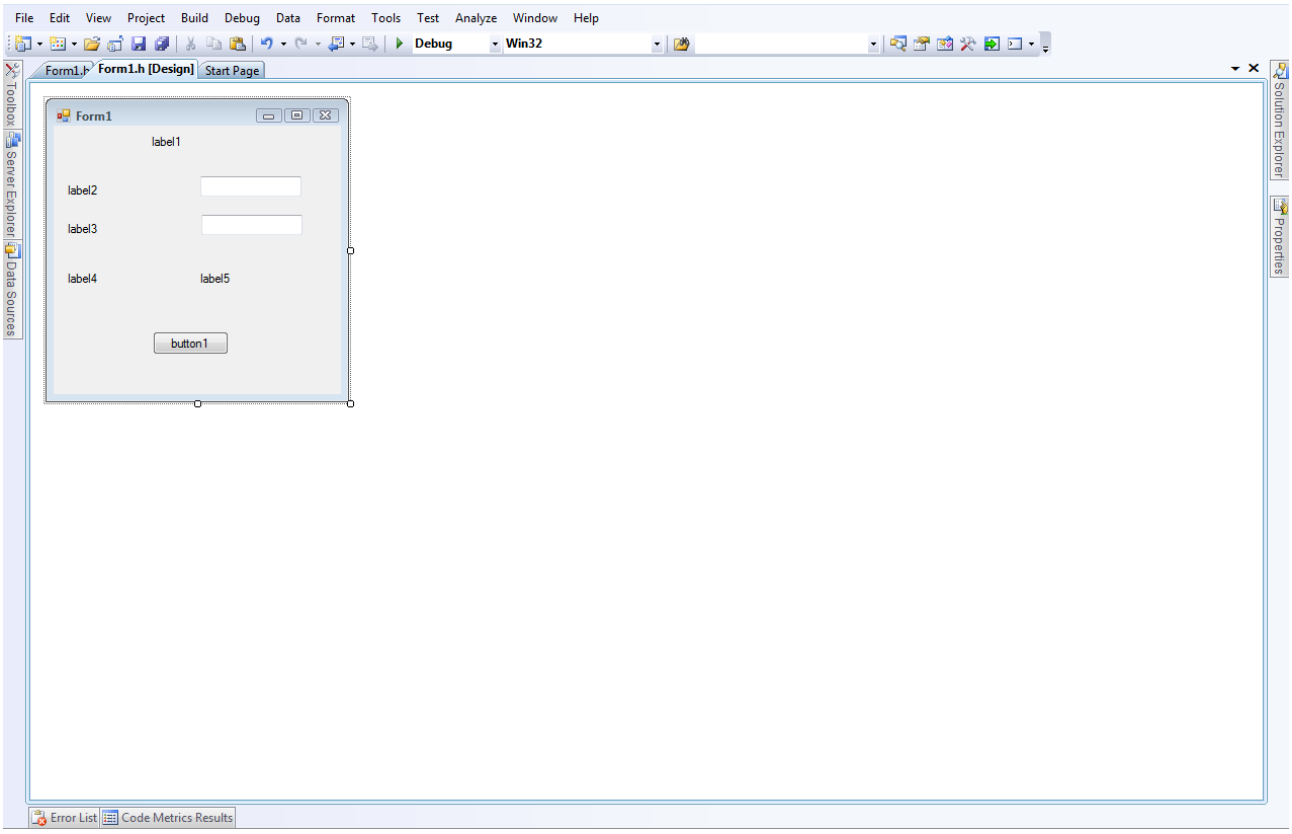
ნახ. 2.2.


File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Win32
Form1.h
Form1.h [Design]*
Start Page
Form1
Toolbox
Server Explorer
Data Sources
Solution Explorer
Properties
Error List
Code Metrics Results

ნახ. 2.3.


File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Win32
Toolbox
All Windows Forms
Common Controls
Pointer
Button
CheckBox
CheckedListBox
ComboBox
DateTimePicker
Label
LinkLabel
ListBox
ListView
MaskedTextBox
MonthCalendar
NotifyIcon
NumericUpDown
PictureBox
ProgressBar
RadioButton
RichTextBox
TextBox
ToolTip
TreeView
WebBrowser
Containers
Pointer
FlowLayoutPanel
GroupBox
Panel
SplitContainer
TabControl
TableLayoutPanel
Menus & Toolbars
Pointer
ContextMenuStrip
MenuStrip
Error List
Code Metrics Results
Ready

ნახ. 2.4.


File Edit View Project Build Debug Data Format Tools Test Analyze Window Help
Debug
Win32
Form1.h
Form1.h [Design]
Start Page
Form1
label1
label2
label3
label4
label5
button1
Error List
Code Metrics Results

ნახ. 2.5.

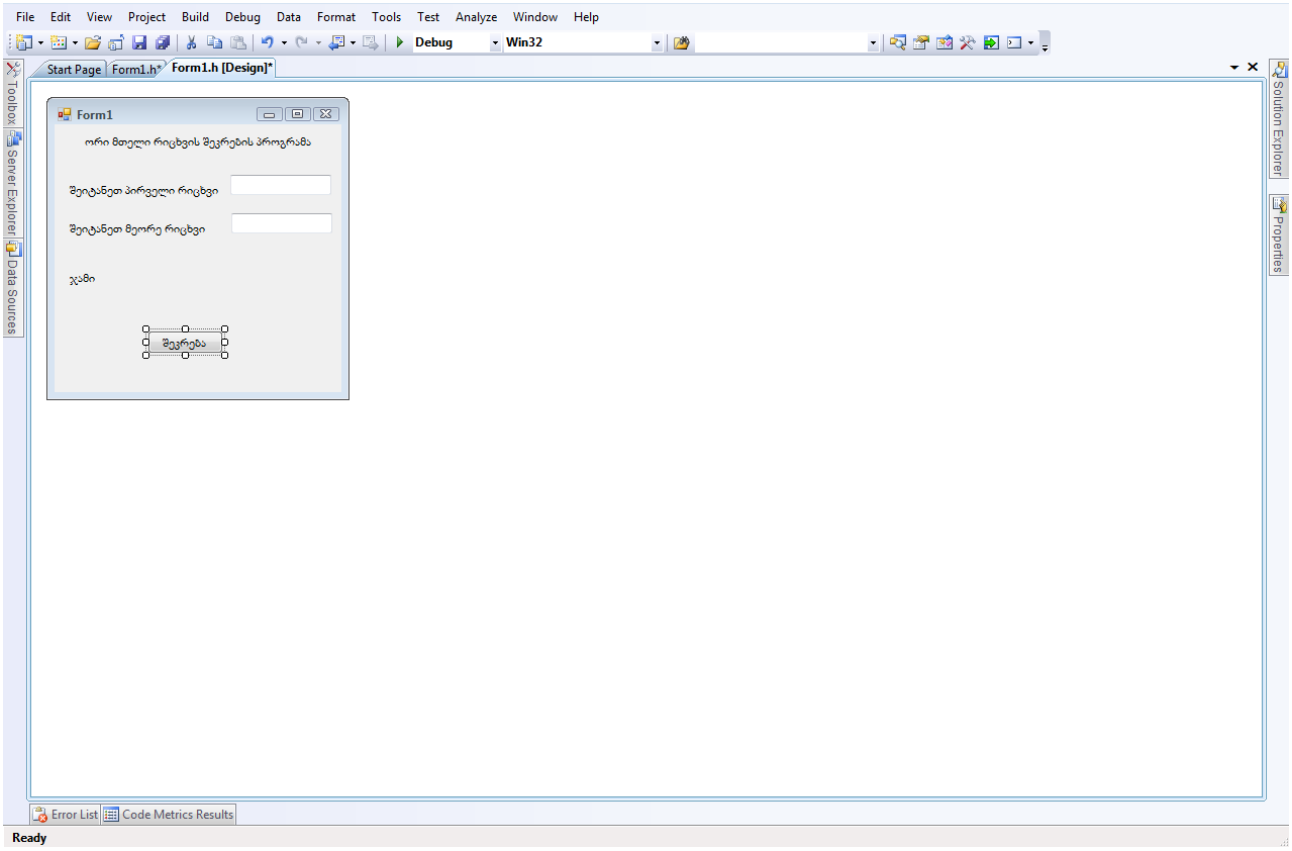
რაც შეეხება უკანასკნელ სტრიქონს, აქ მინიჭების ოპერატორის მარცხნივ მოთავსებულია label5 კომპონენტი. მის Text თვისებაში ვწერთ შედეგს, კერძოდ, jami ცვლადის მნიშვნელობას. რადგან Text თვისებას აქვს სტრიქონული ტიპი, ამიტომ jami ცვლადის ტიპიც უნდა გარდავქმნათ მთელი რიცხვიდან სტრიქონად. სწორედ ამას აკეთებს ToString() ფუნქცია.

ახლა შევასრულოთ ეს პროგრამა. ამისათვის, ჯერ გავუშვათ Microsoft Visual Studio .NET 2008 სისტემა. გაიხსნება Microsoft Development Environment ფანჯარა (ნახ. 2.1). ვაჭერთ Recent Projects განყოფილების Create სიტყვის მარჯვნივ მოთავსებულ Project მიმართვას. გაიხსნება New Project ფანჯარა (ნახ. 2.2). Project Types სიაში მოვნიშნოთ Visual C++ ელემენტი. Templates ზონაში მოვნიშნოთ Windows Forms Application ელემენტი. Name ველში უნდა შევიტანოთ პროექტის სახელი, მაგალითად, OriRicxvisShekreba. შემდეგ ვაჭერთ Browse კლავიშს და მოვნიშნავთ იმ კატალოგს, რომელშიც უნდა შეიქმნას OriRicxvisShekreba ქვეკატალოგი (შეგვიძლია წინასწარ შევქმნათ ის კატალოგი, რომელშიც მოვათავსებთ აღნიშნულ ქვეკატალოგს). ვაჭერთ Open კლავიშს, შემდეგ კი - Ok კლავიშს. თუ კატალოგს არ ავირჩევთ, მაშინ OriRicxvisShekreba ქვეკატალოგი შეიქმნება Visual Studio 2008 კატალოგის Projects ქვეკატალოგში. ეკრანზე გამოჩნდება ფორმა, რომლის სახელია Form1 (ნახ. 2.3). ფორმის მარცხნივ მოთავსებულია ვიზუალური კომპონენტების პანელი (ToolBox). მასზე მოვათავსოთ ისარი (კურსორი, მიმთითებელი). გამოჩნდება ინსტრუმენტების პანელი (ნახ. 2.4). გავხსნათ Common Controls პანელი. ავირჩიოთ textBox კომპონენტი. ამისათვის, მასზე მოვათავსოთ მიმთითებელი, დავაჭიროთ თაგვის მარცხენა კლავიშს, შემდეგ, მიმთითებელი მოვათავსოთ ფორმაზე საჭირო ადგილას და ისევ დავაჭიროთ თაგვის მარცხენა კლავიშს. შედეგად, კომპონენტი ფორმაზე გამოჩნდება მითითებულ ადგილზე. მისი სახელი იქნება textBox1 (ნახ. 2.5). ასეთი გზით ფორმაზე მოვათავსოთ კიდევ ერთი textBox კომპონენტი - textBox2, ერთი Button კომპონენტი და ხუთი label კომპონენტი. ფორმა მიიღებს ნახ. 2.5-ზე ნაჩვენებ სახეს. ვიზუალური კომპონენტები ფორმაზე შეგვიძლია განვათავსოთ ჩვენი მოსაზრებებიდან და გემოვნებიდან გამომდინარე. აქ შემოდის დიზაინის საკითხებიც, კერძოდ, ფორმაზე კომპონენტები ისე უნდა იყოს განლაგებული, რომ მათთან მუშაობა მოხერხებული იყოს.

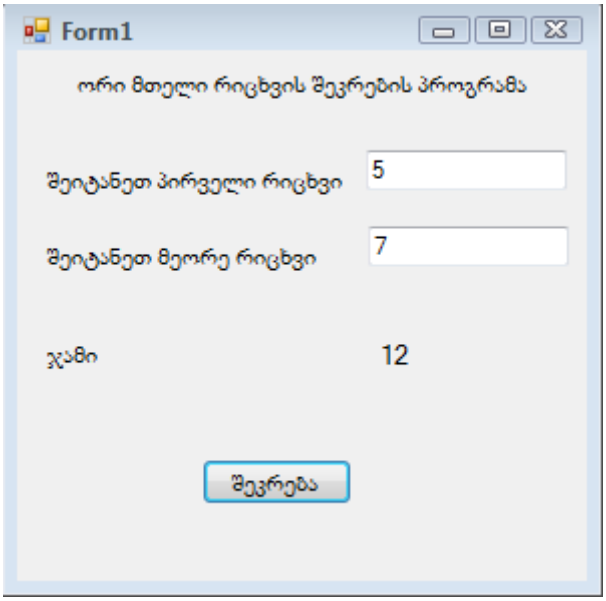
მოვნიშნოთ label1 კომპონენტი და მის Text თვისებაში, რომელსაც ვიპოვით ფანჯრის მარჯვენა მხარეს მოთავსებული თვისებების (Properties) ფანჯარაში, შევიტანოთ ტექსტი: "ორი მთელი რიცხვის შეკრების პროგრამა" და დავაჭიროთ კლავიატურის Enter კლავიშს. შემდეგ, ამავე ფანჯრის Font თვისებაში ვირჩევთ ქართულ შრიფტს, ზომას და სტილს. ანალოგიურად ვიქცევით დანარჩენი კომპონენტებისთვისაც. მივიღებთ ნახ. 2.6-ზე ნაჩვენებ ფორმას.

ფორმაზე კომპონენტების განლაგების შემდეგ ორჯერ სწრაფად ვაწკაპუნებთ button1 კლავიშზე. გაიხსნება პროგრამის კოდის ზონა, რომელშიც შეგვაქვს პროგრამა 2.2. პროგრამის შეტანის შემდეგ მის შესანახად უნდა დავაჭიროთ ინსტრუმენტების პანელის Save All კლავიშს. ფორმაზე გადასასვლელად ვაჭერთ Shift+F7 კლავიშებს (+ ნიშნავს კლავიშებზე ერთდროულ დაჭერას), ფორმიდან პროგრამის კოდზე გადასასვლელად კი - F7 კლავიშს. პროგრამის შესასრულებლად გამოიყენება F5 კლავიში. თუ პროგრამა უშეცდომოდ შევიტანეთ, მაშინ ეკრანზე გამოჩნდება ფორმა (ნახ. 2.7). textBox1 და textBox2 კომპონენტებში შევიტანოთ მთელი რიცხვები და დავაჭიროთ button1 კლავიშს, რომლის სათაურში ჩანს სიტყვა "შეკრება". label5 კომპონენტში გამოჩნდება შეტანილი რიცხვების ჯამი. ამის შემდეგ, კვლავ შეგვიძლია შევიტანოთ სხვა რიცხვები და დავაჭიროთ "შეკრება" კლავიშს და ა.შ. ბოლოს, პროგრამის დასამთავრებლად ვაჭერთ ფორმის ზედა მარჯვენა კუთხეში მოთავსებულ ☒ კლავიშს.

C++ პროგრამის ფაილებს აქვთ .h და .cpp გაფართოება (ტიპი). მაგალითად, იმ ფაილის სახელი, რომელშიც ჩვენი პროგრამის საწყისი კოდი შეგვაქვს, არის Form1.h . თუ ამ ფაილს ყურადღებით დავათვალიერებთ, შევნიშნავთ, რომ პროგრამის კოდის ნაწილს ავტომატურად ქმნის Visual C++.NET სისტემა.



ნახ. 2.6.



ნახ. 2.7.

პროგრამა 2.1-ის შესასრულებლად შეგვიძლია შევქმნათ აგრეთვე, კონსოლური პროგრამა-დანართი. ამისათვის, ნახ. 2.2-ზე ნაჩვენები ფანჯრის Templates ზონაში უნდა მოვნიშნოთ CLR Console Application ელემენტი, Name: ველში შევიტანოთ პროგრამის სახელი, კატალოგი ავირჩიოთ Browse კლავიშის საშუალებით და დავაჭიროთ OK კლავიშს. გახსნილ ფანჯარაში გამოჩნდება კოდი:

// c1.cpp : main project file

```
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
   Console::WriteLine(L"Hello World");
   return 0;
}
```

აქ Console::WriteLine() მეთოდს ეკრანზე გამოაქვს მითითებული სტრიქონი. იმისათვის, რომ ეს სტრიქონი ეკრანზე დავაყოვნოთ, Console::WriteLine() მეთოდის შემდეგ უნდა მოვათავსოთ Console::ReadLine() მეთოდი; შედეგი ეკრანზე გამოტანილი იქნება მანამ, სანამ არ დავაჭერთ Enter კლავიშს. არსებობს ეკრანზე შედეგის დაყოვნების მეორე გზა. ამისათვის, using namespace System; გამოცხადების შემდეგ უნდა მოვათავსოთ დირექტივა:

using namespace System::Threading;

Threading სახელების სივრცეში გამოცხადებულია Sleep() ფუნქცია, რომელსაც გამოვიყენებთ ეკრანზე შედეგის დასაყოვნებლად.

შემდეგ, Console::WriteLine(L"Hello World"); სტრიქონის ნაცვლად უნდა შევიტანოთ პროგრამის კოდი. ზემოთ მოცემული პროგრამა შემდეგ სახეს მიიღებს:

```
//      c1.cpp : main project file
#include "stdafx.h"
using namespace System;
using namespace System::Threading;
int main(array<System::String ^> ^args)
{
int ricxvi1, ricxvi2, jami;
String^ m1;
String^ m2;
//      რიცხვები ცალ-ცალკე სტრიქონში შეგვაქვს
m1 = Console::ReadLine();
m2 = Console::ReadLine();
ricxvi1 = Convert::ToInt32(m1);
ricxvi2 = Convert::ToInt32(m2);
jami = ricxvi1 + ricxvi2;
Console::WriteLine(L"jami = " + jami.ToString());
Thread::Sleep(5000);
return 0;
}
```

პროგრამის გაშვების შემდეგ გამოჩნდება შავი ეკრანი. თითო სტრიქონში უნდა შევიტანოთ თითო რიცხვი. ამის შემდეგ დავინახავთ პასუხს. პროგრამის

Console::WriteLine(L"jami = " + jami.ToString());

ფუნქციაში L გამოიყენება უნიკოდ-სიმბოლოების (Unicode) გამოსატანად. Thread::Sleep(5000); ფუნქცია ეკრანზე შედეგებს აყოვნებს 5 წამის განმავლობაში.

ბოლოს, შევნიშნოთ, რომ C++ ენაში განსხვავებულია ზედა და ქვედა რეგისტრის სიმბოლოები. მაგალითად, განსხვავებულია Computer და computer სახელები. მაგალითი:

```
{
//      C++ ენაში განსხვავებულია ზედა და ქვედა რეგისტრის სიმბოლოები
int ricxvi1;
Ricxvi1 = 5;    //      შეცდომა! Ricxvi1 ცვლადი არ არის გამოცხადებული
```

```
}
```

როგორც მაგალითიდან ჩანს, გამოცხადებული იყო ცვლადი, რომლის სახელია ricxcvi1, ხოლო გამოიყენება სახელი Ricxvi1. C++ ენისთვის ეს ორი სხვადასხვა სახელია.

კომენტარები

C++ ენაში არსებობს ერთსტრიქონიანი და მრავალსტრიქონიანი კომენტარები. ერთსტრიქონიანი კომენტარი // სიმბოლოებით იწყება. მრავალსტრიქონიანი კომენტარი /* სიმბოლოებით იწყება და */ სიმბოლოებით მთავრდება. როგორც წესი, კომენტარები შეიცავს პროგრამის სხვადასხვა ნაწილის განმარტებას, ცვლადებისა და ფუნქციების დანიშნულებას და ა.შ. კომენტარების გამოყენება აადვილებს პროგრამის გარჩევას. პროგრამის შესრულებისას ხდება კომენტარების გამოტოვება. ქვემოთ მოცემულია პროგრამა, რომელიც ორივე სახის კომენტარს შეიცავს.

```
{
/*
ეს არის ჩვენი                                    ეს არის მრავალსტრიქონიანი კომენტარი
პირველი პროგრამა
*/
int ricxvi1, ricxvi2, jami;
//      რიცხვების შეტანა textBox კომპონენტებიდან      ეს არის ერთსტრიქონიანი კომენტარი
ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
jami = ricxvi1 + ricxvi2;
label1->Text = jami.ToString();
}
```

C++ ენის საბაზო ტიპები

ინფორმაციის სახესხვაობას, რომელსაც შეძლება ცვლადი შეიცავდეს, ***მონაცემთა ტიპი*** ეწოდება. პროგრამის მონაცემები, ცვლადები და მუდმივები მონაცემთა გარკვეულ ტიპს უნდა ეკუთვნოდეს. ISO/ANSI C++ სტანდარტში განსაზღვრულია ***მონაცემების საბაზო ტიპები.*** C++/CLI სტანდარტშიც განსაზღვრულია მონაცემთა საბაზო ტიპები, რომლებიც არ არიან ISO/ANSI სტანდარტის ნაწილი. მონაცემთა საბაზო ტიპები სამ ნაწილად იყოფა:

- მთელი რიცხვების შემცველი ტიპები;
- ტიპები, რომლებიც შეიცავენ არამთელრიცხვა მნიშვნელობებს;
- void ტიპი, რომელიც მიუთითებს მნიშვნელობების ცარიელ სიმრავლეზე ან ტიპის არარსებობაზე.

ISO/ANSI C++ ტიპები მოცემულია 2.1 ცხრილში. C++/CLI ტიპები მოცემულია 2.2 ცხრილში. გარდა ამისა, System სახელების სივრცე შეიცავს კლასებს, რომლებიც მონაცემთა საბაზო ტიპებია და რომლებსაც იყენებენ C#, C++, Visual Basic და JScript ენებზე შედგენილი პროგრამები. System არის საბაზო სახელების სივრცე .NET Framework გარემოსთვის. მასში განსაზღვრული ტიპები მოცემულია 2.3 ცხრილში.

C++/CLI პროგრამაში საბაზო ტიპი არის კლასის ტიპის მქონე მნიშვნელობა და შეიძლება იქცეოდეს როგორც ჩვეულებრივი მნიშვნელობა ან როგორც ობიექტი.

C++/CLI ენაში თითოეული ISO/ANSI საბაზო ტიპი აისახება ***მნიშვნელობის ტიპის კლასზე***, რომელიც განსაზღვრულია System სახელების სივრცეში. ამრიგად, C++/CLI პროგრამაში

ISO/ANSI საბაზო ტიპის სახელებია მათთან ასოცირებული ტიპების კლასების შემოკლებული სახელები. ამიტომ, შეგვიძლია საბაზო ტიპის მნიშვნელობა განვიხილოთ როგორც უშუალოდ მნიშვნელობა ან როგორც ასოცირებული ტიპის კლასის ავტომატურად გარდაქმნილი ობიექტი. საბაზო ტიპები, მათ მიერ დაკავებული მეხსიერების ზომა და მათი შესაბამისი მნიშვნელობების ტიპების კლასები მოცემულია 2.2 ცხრილში.

ავტომატურად (default), char ტიპი signed char ტიპის ეკვივალენტურია. ამიტომ, მასთან ასოცირებულია System::SByte მნიშვნელობის ტიპის კლასი. System არის საწყისი სახელების სივრცე, რომელშიც C++/CLI მნიშვნელობების ტიპების კლასებია განსაზღვრული. System სახელების სივრცეში განსაზღვრულია ბევრი სხვა ტიპი. მაგალითად, Decimal ტიპის ცვლადი, რომელიც 28 ათობითი თანრიგისაგან შედგება.

საწყისი მნიშვნელობის გარდაქმნას ასოცირებული ტიპის კლასად ***შეფუთვა*** (boxing, упаковка) ეწოდება. პირიქით გარდაქმნას კი - ***განფუთვა*** (unboxing, распаковка). ეს, ამ ტიპის ცვლადებს საშუალებას აძლევს მოიქცნენ როგორც მარტივი ცვლადები ან როგორც ობიექტები.

C++/CLI პროგრამაში ყველა მონაცემი ინახება როგორც მნიშვნელობების ტიპების, ისე მიმართვითი ტიპების კლასების ობიექტების სახით.

მთელრიცხვა ტიპები

მთელრიცხვა ცვლადები მხოლოდ მთელ რიცხვებს ინახავენ. C++ ISO/ANSI და C++/CLI მთელრიცხვა ტიპები მოცემულია 2.1 და 2.2 ცხრილებში. აქედან unsigned მოდიფიკატორიანი ტიპები გამოიყენება უნიშნო მთელი რიცხვების წარმოსადგენად, დანარჩენი ტიპები კი - ნიშნიანი რიცხვების წარმოსადგენად. განსხვავება ნიშნიან და უნიშნო მთელ რიცხვებს შორის შემდეგშია. ნიშნიან რიცხვებში ნიშნის მისათითებლად უფროსი ბიტი (ოროობითი თანრიგი) გამოიყენება. თუ ეს ბიტი ნულის ტოლია, რიცხვი დადებითია, თუ ერთის ტოლია, რიცხვი უარყოფითია. უნიშნო რიცხვებში ნიშნის წარმოსადგენად უფროსი ბიტი არ გამოიყენება და შედის რიცხვის შედგენილობაში. ამიტომ, უნიშნო რიცხვების აბსოლუტური მნიშვნელობა ორჯერ აღემატება ნიშნიანი რიცხვის მნიშვნელობას.

C++/CLI ვერსიაში განსაზღვრულია ორი ახალი მთელი ტიპი (2.2 ცხრილი): long long და unsigned long long. long long ტიპის მნიშვნელობების დიაპაზონია -9,223,372,036,854,775,808 ÷ 9,223,372,036,854,775,807 , unsigned long long ტიპის კი - 0 ÷ 18,446,744,073,709,551,615 .

long long ტიპის ლიტერალის განსაზღვრისათვის უნდა გამოვიყენოთ LL ან ll სუფიქსი, unsigned long long ტიპის ლიტერალის განსაზღვრისათვის კი - ULL ან ull სუფიქსი:

```
long long mteli1 = 123456789123456LL;
unsigned long long mteli2 = 12345678912345678ULL;
```

ISO/ANSI პროგრამაში შეგვიძლია გამოვიყენოთ მხოლოდ ISO/ANSI მთელრიცხვა ტიპები. C++/CLI პროგრამაში შეგვიძლია გამოვიყენოთ როგორც ISO/ANSI, ისე CLI ტიპები:

```
{
// პროგრამით ხდება მთელ რიცხვებთან მუშაობის დემონსტრირება
int mteli1 = Convert::ToInt32(textBox1->Text);
double wiladi1 = Convert::ToDouble(textBox2->Text);
Int32 mteli2 = Convert::ToInt32(textBox3->Text);
Double wiladi2 = Convert::ToDouble(textBox4->Text);
Byte baiti = Convert::ToSByte(textBox5->Text);
long long mteli3 = Convert::ToInt64(textBox6->Text);
label1->Text = mteli1.ToString() + "  " + wiladi1.ToString() + "  " + mteli2.ToString() + "  " +
                wiladi2.ToString() + "  " + baiti.ToString() + "  " + mteli3.ToString();
}
```

ცხრილი 2.1. C++ ISO/ANSI ტიპები

ტიპი	აღწერა	ბაიტების რაოდენობა	მნიშვნელობების დიაპაზონი
bool	ლოგიკური მნიშვნელობა	1	false ან true
char	ნიშნიანი სიმბოლო	1	–128÷ 127
signed char	ნიშნიანი სიმბოლო	1	–128 ÷ 127
unsigned char	უნიშნო სომბოლო	1	0 ÷ 255
wchar_t	ორბაიტიანი char	2	0 ÷ 65,535
short	ნიშნიანი მოკლე მთელი რიცხვი	2	–32,768 ÷ 32,767
unsigned short	უნიშნო მოკლე მთელი რიცხვი	2	0 ÷ 65,535
long	ნიშნიანი გრძელი მთელი რიცხვი	4	–2,147,483,648 ÷ 2,147,483,647
int	ნიშნიანი მთელი რიცხვი	4	–2,147,483,648 ÷ 2,147,483,647
unsigned int	უნიშნო მთელი რიცხვი	4	0 ÷ 4,294,967,295
unsigned long	უნიშნო გრძელი მთელი რიცხვი	4	0 ÷ 4,294,967,295
float	ნიშნიანი წილადი	4	3.4E +/- 38 (7 ციფრი)
double	ნიშნიანი წილადი	8	1.7E +/- 308 (15 ციფრი)
long double	ნიშნიანი გრძელი წილადი	იგივეა რაც double	იგივეა რაც double
__int8	8 ბიტიანი ნიშნიანი მთელი რიცხვი	1	–128 ÷ 127
unsigned __int8	8 ბიტიანი უნიშნო მთელი რიცხვი	1	0 ÷ 255
__int16	ნიშნიანი მოკლე მთელი რიცხვი	2	–32,768 ÷ 32,767
unsigned __int16	უნიშნო მოკლე მთელი რიცხვი	2	0 ÷ 65,535
__int32	ნიშნიანი მთელი რიცხვი	4	–2,147,483,648 ÷ 2,147,483,647
unsigned __int32	უნიშნო მთელი რიცხვი	4	0 ÷ 4,294,967,295
__int64	ნიშნიანი გრძელი მთელი რიცხვი	8	–9,223,372,036,854,775,808÷ 9,223,372,036,854,775,807
unsigned __int64	უნიშნო გრძელი მთელი რიცხვი	8	0 ÷ 18,446,744,073,709,551,615

ცხრილი 2.2. C++/ CLI ტიპები

CLI მნიშვნელობის ტიპის კლასი	საბაზო ტიპი	ზომა ბაიტებში
System::Boolean	bool	1
System::SByte	char	1
System:: SByte	signed char	1
System:: Byte	unsigned char	1
System::Int16	short	2
System:: UInt16	unsigned short	2
System::Int32	int	4
System::UInt32	unsigned int	4
System:: Int32	long	4
System:: UInt32	unsigned long	4
System::Int64	long long	8
System::UInt64	unsigned long long	8
System::Single	float	4
System::Double	double	8
System::Double	long double	8
System::Char	wchar_t	2

მთელრიცხვა ტიპების მოდიფიკატორები

char, short, int და long ტიპის ცვლადებისთვის იგულისხმება signed ტიპის მოდიფიკატორი. ეს იმას ნიშნავს, რომ ისინი ინახავენ ნიშნიან მთელ მნიშვნელობებს ანუ დადებით და უარყოფით მნიშვნელობებს. მაგალითად, როდესაც ვწერთ int ან long ტიპს, იგულისხმება signed int ან signed long ტიპი.

თუ ცვლადი იღებს მხოლოდ დადებით მნიშვნელობებს, მაშინ მისი გამოცხადებისას შეგვიძლია მივუთითოთ unsigned მოდიფიკატორი:

```
unsigned int mteli = 87;
```

თუ ვიყენებთ მხოლოდ signed ან unsigned მოდიფიკატორს, მაშინ შესაბამისად, იგულისხმება signed int ან unsigned int:

```
signed mteli1 = 75;          //   იგულისხმება signed int mteli1 = 75;
unsigned mteli2 = 75;        //   იგულისხმება unsigned int mteli2 = 75;
```

მოძრავწერტილიანი ტიპები

მოძრავწერტილიანი ცვლადები წილად რიცხვებს ინახავენ. პროგრამაში შეგვიძლია გამოვიყენოთ როგორც ISO/ANSI, ისე C++/CLI ტიპები. ქვემოთ მოცემულ პროგრამაში ხდება წილად რიცხვებთან მუშაობის დემონსტრირება.

ცხრილი 2.3. System სახელების სივრცეში განსაზღვრული მონაცემთა ტიპები

კატეგორია	კლასის სახელი	აღწერა	Visual Basic მონაცემთა ტიპი	C# მონაცემთა ტიპი	C++ მონაცემთა ტიპი	JScript მონაცემთა ტიპი
მთელი	Byte	8-ბიტიანი უნიშნო მთელი	Byte	byte	char	Byte
	SByte	8-ბიტიანი ნიშნიანი მთელი	SByte	sbyte	signed char	SByte
	Int16	16-ბიტიანი ნიშნიანი მთელი	Short	short	short	short
	Int32	32-ბიტიანი ნიშნიანი მთელი	Integer	int	int ან long	int
	Int64	64-ბიტიანი ნიშნიანი მთელი	Long	long	__int64	long
	UInt16	16-ბიტიანი უნიშნო მთელი	UShort	ushort	unsigned short	UInt16
	UInt32	32-ბიტიანი უნიშნო მთელი	UInteger	uint	unsigned int ან unsigned long	UInt32
	UInt64	64-ბიტიანი უნიშნო მთელი	ULong	ulong	unsigned __int64	UInt64
მცურავი წერტილი	Single	ერთმაგი სიზუსტის (32-ბიტი) მოძრავ-წერტილიანი რიცხვი	Single	float	float	float
	Double	ორმაგი სიზუსტის (64-ბიტი) მოძრავ-წერტილიანი რიცხვი	Double	double	double	double
ლოგიკური	Boolean	Boolean მნიშვნელობა (true ან false)	Boolean	bool	bool	bool

ცხრილი 2.3. (გაგრძელება)

სხვა	Char	უნიკოდ-სიმბოლო (16-ბიტი)	Char	char	wchar_t	char
	Decimal	Decimal სიდიდე (128-ბიტი)	Decimal	decimal	Decimal	Decimal
	IntPtr	ნიშნიანი მთელი რიცხვი, რომლის ზომა დამოკიდებულია ლატფორმაზე (32-ბიტი ან 64-ბიტი)	IntPtr (არაჩადგმული ტიპი)	IntPtr (არაჩადგმული ტიპი)	IntPtr (არაჩადგმული ტიპი)	IntPtr
	UIntPtr	უნიშნო მთელი რიცხვი, რომლის ზომა დამოკიდებულია პლატფორმაზე (32-ბიტი ან 64-ბიტი)	UIntPtr (არაჩადგმული ტიპი)	UIntPtr (არაჩადგმული ტიპი)	UIntPtr (არაჩადგმული ტიპი)	UIntPtr
კლასის ობიექტები	Object	ობიექტების იერარქიის ფუძე	Object	object	Object*	Object
	String	უნიკოდ-სიმბოლოების ფიქსირებული სიგრძის უცვლელი სტრიქონი	String	string	String*	String

```
{
//          პროგრამაში ხდება წილადებთან მუშაობის დემონსტრირება
float    wiladi1, wiladi2, shedegi1;
double wiladi3, wiladi4, shedegi2;
Double wiladi5, wiladi6, shedegi3;

wiladi1 = 5.5;
wiladi3 = 7.98;
wiladi5 = 3.084;
wiladi2 = Convert::ToDouble(textBox1->Text);
wiladi4 = Convert::ToDouble(textBox2->Text);
wiladi6 = Convert::ToDouble(textBox3->Text);
shedegi1 = wiladi1 + wiladi2;
```

```
shedegi2 = wiladi3 + wiladi4;
shedegi3 = wiladi5 + wiladi6;
label1->Text = shedegi1.ToString() + "  " + shedegi2.ToString() + "  " + shedegi3.ToString();
}
```

მოძრავწერტილიანი მუდმივების განსაზღვრისას აუცილებლად უნდა მივუთითოთ წერტილი ან ექსპონენტა, ან ორივე ერთად. წინააღმდეგ შემთხვევაში, მივიღებთ მთელ რიცხვს:

```
const double d1= 9.43;          //      d1 = 9.43
double d2 = 5.6E-3;             //      d2 = 0,0056
double d3 = 6E3;                //      d3 = 6000
```

სიმბოლური ტიპები

სიმბოლური ცვლადები სიმბოლოებს ინახავენ. პროგრამაში შეგვიძლია გამოვიყენოთ როგორც ISO/ANSI, ისე C++/CLI სიმბოლური ტიპები. char ტიპი ერთ ბაიტს იკავებს, wchar_t და Char კი - 2 ბაიტს. ამიტომ, wchar_t და Char ტიპები გამოიყენება Unicode სიმბოლოებთან სამუშაოდ. მაგალითი.

```
{
//      პროგრამაში ხდება სიმბოლოებთან მუშაობის დემონსტრირება
char simbolo1 = 'w';
char simbolo2 = textBox1->Text[0];
wchar_t simbolo3 = L'რ';
wchar_t simbolo4 = textBox2->Text[0];
Char simbolo5 = L'დ';
Char simbolo6 = textBox3->Text[0];

label1->Text = simbolo1.ToString() + "   " + simbolo2.ToString();
label2->Text = simbolo3.ToString() + "   " + simbolo4.ToString();
label3->Text = simbolo5.ToString() + "   " + simbolo6.ToString();
}
```

სიმბოლურ ცვლადებს შეგვიძლია, აგრეთვე, მივანიჭოთ მმართველი სიმბოლოები. ***მმართველია სიმბოლო***, რომელიც გამოიყენება გარკვეული მოქმედებების შესასრულებლად. ეს მოქმედებებია: ახალ სტრიქონზე გადასვლა, ჰორიზონტალური ტაბულირება, სტრიქონის დასაწყისში გადასვლა და ა.შ. მაგალითად,

```
label1->Text = L"საბა \n ანა";
```

მინიჭების შესრულების შედეგად label1 კომპონენტის პირველ სტრიქონში გამოჩნდება "საბა", მეორე სტრიქონში კი - "ანა". 2.4 ცხრილში მოცემულია მმართველი სიმბოლოები.

მმართველი სიმბოლოების გამოყენების შედეგები უფრო მკაფიოდ ჩანს კონსოლურ პროგრამა-დანართებთან მუშაობისას. საბრძანებო სტრიქონის სიმბოლოზე ორიენტირებულ პროგრამას **კონსოლური პროგრამა** ეწოდება. იგი ჩვენთან ურთიერთქმედებს კლავიატურისა და ეკრანის საშუალებით, რომლებიც სიმბოლურ რეჟიმში მუშაობენ.

არსებობს კონსოლური პროგრამების ორი სახე: Win32 კონსოლური პროგრამები, რომლებიც კომპილირდება მშობლიურ კოდში, და CLR კონსოლური პროგრამები, რომლებიც ორიენტირებულია CLR გარემოში შესრულებაზე.

მაგალითი.

ცხრილი 2.4. მმართველი სიმბოლოები

მმართველი სიმბოლო	აღწერა
\'	ერთმაგი ბრჭყალი
\"	ორმაგი ბრჭყალი
\\	ირიბი დახრილი ხაზი
\0	Null (სიმბოლო, რომლის კოდია 0)
\a	ზარი
\b	უკან დაბრუნება (BackSpace)
\f	გვერდის გადაფურცვლა
\n	ახალ სტრიქონზე გადასვლა
\r	სტრიქონის დასაწყისში გადასვლა
\t	ჰორიზონტალური ტაბულირება
\v	ვერტიკალური ტაბულირება

```
// პროგრამაში ხდება მმართველ სიმბოლოებთან მუშაობის დემონსტრირება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

int main(array<System::String ^> ^args)
{
Console::WriteLine(L"რომანი\tლიკა\tბექა");
Console::WriteLine(L"ანა\nსაბა\a");
Console::WriteLine(L"რომანი\bლიკა");
Console::WriteLine(L"რომანი ლიკა\rანა");
Thread::Sleep(5000);
return 0;
}
```

როგორც პროგრამის შესრულების შედეგებიდან ჩანს:

```
romani	lika	beqa
ana
saba
romanlika
anaani lika
```

Console::WriteLine(L"რომანი\tლიკა\tბექა"); ფუნქციის მუშაობის შედეგად, შესრულდება გამოსატანი სტრიქონის ჰორიზონტალური ტაბულირება. "რომანი" სტრიქონი გამოჩნდება პირველი პოზიციიდან. შემდეგ, სრულდება ჰორიზონტალური ტაბულირება. შედეგად, გამოიტოვება პირველი რვა პოზიცია და მომდევნო "ლიკა" სტრიქონი გამოჩნდება მე-9 პოზიციიდან. შემდეგ, ისევ შესრულდება ჰორიზონტალური ტაბულირება, ანუ გამოიტოვება მომდევნო რვა პოზიცია და "ბექა" სტრიქონი გამოჩნდება მე-17 პოზიციიდან. ბოლოს WriteLine() ფუნქცია შეასრულებს მომდევნო, მეორე სტრიქონზე გადასვლას. არსებობს, აგრეთვე, Write() ფუნქცია, რომელიც სტრიქონის გამოტანის შემდეგ არ ახდენს მომდევნო სტრიქონზე გადასვლას.

Console::WriteLine(L"ანა\nსაბა\a"); ფუნქციის შესრულების შედეგად ეკრანის მეორე

სტრიქონში გამოჩნდება "ანა" სტრიქონი. შემდეგ, მოხდება მომდევნო, მესამე სტრიქონზე გადასვლა, რადგან ამ სტრიქონს მოსდევს მომდევნო სტრიქონზე გადასვლის სიმბოლო. მესამე სტრიქონში გამოჩნდება "საბა" სტრიქონი და გავიგონებთ კომპიუტერის დინამიკის ხმას.

Console::WriteLine(L"რომანი\ხლიკა"); ფუნქციის შესრულების შედეგად "რომანი" სტრიქონის უკანასკნელი სიმბოლო წაიშლება და შემდეგ გამოჩნდება "ლიკა" სტრიქონი.

Console::WriteLine("რომანი ლიკა\rანა"); ფუნქციის შესრულების შედეგად ჯერ გამოჩნდება "რომანი ლიკა" სტრიქონი, შემდეგ მოხდება ამავე სტრიქონის დასაწყისში გადასვლა და "ანა" სტრიქონის გამოტანა. შედეგად, მიიღება "ანაანი ლიკა" სტრიქონი.

bool ტიპი

ბულის ტიპის ცვლადი (ლოგიკური ცვლადი) იღებს ორ მნიშვნელობას: **true** (ჭეშმარიტი) და **false** (მცდარი). მაგალითი.

```
{
//      პროგრამაში ხდება ლოგიკურ ცვლადთან მუშაობის დემონსტრირება
bool b1, b2;

b1 = true;
b2 = Convert::ToBoolean(textBox1->Text);
label1->Text = b1.ToString();
label2->Text = b2.ToString();
}
```

ToBoolean ფუნქცია textBox1 კომპონენტში შეტანილ სტრიქონს გარდაქმნის ლოგიკურ მონაცემად. ამ შემთხვევაში, textBox1 კომპონენტში უნდა შევიტანოთ true ან false.

String^ ტიპი

სტრიქონებთან სამუშაოდ შეგვიძლია გამოვიყენოთ System სახელების სივრცეში განსაზღვრული String^ ტიპი. მაგალითი:

```
{
//      პროგრამაში ხდება სტრიქონებთან მუშაობის დემონსტრირება
String^ striqoni1 = L"ანა და საბა";
String^ striqoni2 = textBox1->Text;
label1->Text = striqoni1;
striqoni1 = striqoni2;
label2->Text = striqoni1;
}
```

ლიტერალები

ნებისმიერი სახის ფიქსირებულ მნიშვნელობას ***ლიტერალი*** ეწოდება. ლიტერალი (2.5 ცხრილი) შეიძლება იყოს მთელი რიცხვი, წილადი, სტრიქონი, სიმბოლო ან ლოგიკური მნიშვნელობა. მაგალითი:

```
{
int mteli1 = -4;                    //      ლიტერალია - -4
long mteli2 = 45L;                  //      ლიტერალია - 45L
```

```
double wiladi = 9.21;                                   //      ლიტერალია - 9.21
bool logikuri = true;                                   //      ლიტერალია - true
Char simbolo = L'რ';                                    //      ლიტერალია - L'რ'
String^ striqoni = L"საბა სამხარაძე";                   //      ლიტერალია - L"საბა სამხარაძე"
label1->Text = mteli1.ToString() + "   " + mteli2.ToString() + "   " +
                wiladi.ToString() + "   " + logikuri.ToString() + "   " + simbolo + "   " + striqoni;
}
```

ცხრილი 2.5. სხვადასხვა ტიპის ლიტერალი

ტიპი	მაგალითები
char, signed char, unsigned char	'A', 'S', '#', '5'
Char, wchar_t	L'რ', L'ს', L'#', L'5'
int	-54, 29713, 0x8FC
unsigned int	25U, 32900U
long	-54L, 29713L
unsigned long	4UL, 987654321UL
float	8.03f
double	6.32
long double	1.73L
bool	true, false

მონაცემთა ტიპისათვის სინონიმის განსაზღვრა

მონაცემთა არსებულ ტიპს შეგვიძლია დავარქვათ ჩვენთვის საჭირო სახელი. ამისათვის გამოიყენება საკვანძო სიტყვა **typedef**. მისი სინტაქსია

typedef ***ტიპი სინონიმი***

საკვანძო სიტყვა typedef შეგვიძლია მივუთითოთ ფაილის დასაწყისში #pragma once დირექტივის შემდეგ:

```
#pragma once
typedef unsigned int Ricxvebi;
```

ან/და

```
"private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {"
```

სტრიქონის წინ:

```
typedef  long long Didi_Mteli;
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Ricxvebi mteli1 = 9;
Didi_Mteli mteli2 = 795LL;
label1->Text = mteli1.ToString() + "   " + mteli2.ToString();
}
```

აქ, ფაილის დასაწყისში, იქმნება unsigned int ტიპის ფსევდონიმი - Ricxvebi. შემდეგ ის გამოიყენება პროგრამაში unsigned int ტიპის ნაცვლად. ანალოგიურად ვქმნით და ვიყენებთ Didi_Mteli ფსევდონიმს.

ხშირად ფსევდონიმების გამოყენება ამარტივებს რთულ გამოცხადებებს ერთი სახელის გამოყენების გზით და აუმჯობესებს საწყისი კოდის წაკითხვას.

მუდმივა

მუდმივა (კონსტანტა, constant) არის ფიქსირებული მნიშვნელობა. მისი გამოცხადება იწყება **const** სიტყვით. მუდმივა შეიძლება იყოს ნებისმიერი ტიპის მქონე მნიშვნელობა, მაგალითად მთელი რიცხვები: 5, -10; მცურავწერტილიანი რიცხვები: 23.54, 1.09; სიმბოლოები: 'Q', 'რ'; სტრიქონი: "C++ დაპროგრამების ენაა" და ა.შ. მოცემული პროგრამით ხდება მუდმივასთან მუშაობის დემონსტრირება.

```
{
//      პროგრამით ხდება მუდმივასთან მუშაობის დემონსტრირება
//      მუდმივას განსაზღვრა
const int mteli_mudmiva = 1024;
const Char simbolo_mudmiva = L'ა';
const float wiladi_mudmiva1 =  12.09;
const double wiladi_mudmiva2 = 456.302;
int cvladi, jami;
double shedegi;

cvladi = Convert::ToInt32(textBox1->Text);
jami = cvladi + mteli_mudmiva;
shedegi = wiladi_mudmiva1 + wiladi_mudmiva2;
label1->Text = jami.ToString();
label2->Text = simbolo_mudmiva.ToString();
label3->Text = wiladi_mudmiva1.ToString();
label4->Text = Math::Round(shedegi,2).ToString();
}
```

ცვლადი და მისი ინიციალიზება

ცვლადი (variable) არის მეხსიერების სახელდებული უბანი, რომელსაც მნიშვნელობა ენიჭება. ეს მნიშვნელობა შეიძლება შეიცვალოს პროგრამის მუშაობის პროცესში. ოპერატორს, რომლის საშუალებითაც ხდება ცვლადის გამოცხადება, შემდეგი სინტაქსი აქვს:

ტიპი ცვლადის_სახელი;

სადაც, ***ტიპი*** ცვლადის ტიპია, ***ცვლადის_სახელი*** - კი მისი სახელი. ნებისმიერი ცვლადი უნდა გამოცხადდეს გამოყენებამდე. წინააღმდეგ შემთხვევაში მივიღებთ შეცდომას. ამასთან, ცვლადს უნდა მიენიჭოს მხოლოდ შესაბამისი ტიპის მნიშვნელობა. მაგალითად, bool ტიპის ცვლადს უნდა მიენიჭოს true ან false მნიშვნელობა და არა წილადი ან სხვა.

ცვლადის ინიციალიზება

ცვლადის გამოცხადებისას მისთვის მნიშვნელობის მინიჭებას ცვლადის ***ინიციალიზება*** ეწოდება. ცვლადის ინიციალიზების ოპერატორის სინტაქსია:

ტიპი ცვლადის_სახელი = მნიშვნელობა;

მნიშვნელობის ტიპი უნდა ემთხვეოდეს ცვლადის ტიპს. როგორც აღვნიშნეთ, ცვლადს მნიშვნელობა უნდა მივანიჭოთ მის გამოყენებამდე. ამის გაკეთება შეიძლება ცვლადის

გამოცხადებისას ან გამოცხადების შემდეგ. ამ პროგრამით ხდება ცვლადების ინიციალიზების დემონსტრირება.

```
{
//        პროგრამით ხდება ცვლადების ინიციალიზების დემონსტრირება
int ricxvi3, ricxvi1 = 25, ricxvi2 = 30;
Char simbolo = L'რ';
double wiladi1 = 150.75;
label2->Text = ricxvi1.ToString() + "   " + wiladi1.ToString() + "   " + simbolo;
}
```

ცვლადების დინამიკური ინიციალიზება

ცვლადების ინიციალიზება შესაძლებელია, აგრეთვე, დინამიკურად, ე.ი. პროგრამის შესრულებისას. ამ შემთხვევაში ცვლადის ინიციალიზება ხდება არა პროგრამის დასაწყისში, არამედ მის ნებისმიერ ადგილას. მაგალითი:

```
{
//        პროგრამით ხდება ცვლადის დინამიკური ინიციალიზების დემონსტრირება
int simagle = 5, sigane, sigrdze;

sigane  = Convert::ToInt32(textBox1->Text);
sigrdze = Convert::ToInt32(textBox2->Text);
// moculoba ცვლადის ინიციალიზება  ხდება დინამიკურად
int moculoba = simagle * sigane * sigrdze;
label1->Text = moculoba.ToString();
}
```

ოპერატორი

ოპერატორი არის სიმბოლო, რომელიც კომპილატორს ატყობინებს თუ რა ოპერაცია უნდა შესრულდეს. C++ ენაში არსებობს ოპერატორების ოთხი ძირითადი კლასი: არითმეტიკის, ლოგიკის, შედარებისა და ბიტობრივი.

არითმეტიკის ოპერატორები

არითმეტიკის ოპერატორები (2.6 ცხრილი) შეგვიძლია გამოვიყენოთ როგორც მთელი, ისე წილადი რიცხვისათვის.

მოვიყვანოთ ზოგიერთი განმარტება. როდესაც გაყოფის ოპერატორს ვიყენებთ მთელი რიცხვებისათვის, მაშინ შედეგი იქნება მთელი რიცხვი, ნაშთი კი უარიყოფა. მაგალითად, 20/6 გაყოფის შედეგი იქნება მთელი რიცხვი 3. ნაშთის მისაღებად უნდა გამოვიყენოთ % ოპერატორი. მაგალითად, 20%6 ოპერაციის შედეგი იქნება 2. % ოპერატორის გამოყენებით შეგვიძლია დავადგინოთ კენტია რიცხვი თუ - ლუწი. მაგალითად, R%2 გამოსახულების გამოთვლისას თუ ნაშთი 1-ის ტოლია, მაშინ R რიცხვი კენტია, თუ ნაშთი 0-ის ტოლია, მაშინ R რიცხვი ლუწია. ანალოგიურად, შეგვიძლია დავადგინოთ ერთი რიცხვი მეორის ჯერადია თუ არა.

ცხრილი 2.6. არითმეტიკის ოპერატორები

ოპერატორი	მნიშვნელობა
+	შეკრება
-	გამოკლება (და უნარული მინუსი)
*	გამრავლება
/	გაყოფა
%	მოდულით გაყოფა (ნაშთის გამოყოფა)
++	ინკრემენტი
--	დეკრემენტი

მოცემული პროგრამებით ხდება არითმეტიკის ოპერატორების გამოყენების დემონსტრირება მთელი რიცხვებისათვის.

```
{
//      პროგრამით ხდება არითმეტიკის ოპერაციების მუშაობის
//      დემონსტრირება მთელი რიცხვებისათვის
int ricxvi1, i2, jami, sxvaoba, namravli, ganayofi, nashti;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
jami      = ricxvi1 + ricxvi2;
sxvaoba  = ricxvi1 - ricxvi2;
namravli = ricxvi1 * ricxvi2;
ganayofi = ricxvi1 / ricxvi2;
nashti    = ricxvi1 % ricxvi2;
label1->Text = jami.ToString();
label2->Text = sxvaoba.ToString();
label3->Text = namravli.ToString();
label4->Text = ganayofi.ToString();
label5->Text = nashti.ToString();
}
```

ქვემოთ მოცემული პროგრამით ხდება არითმეტიკის ოპერატორების მუშაობის დემონ-სტრირება წილადი რიცხვებისათვის.

```
{
//      პროგრამით ხდება არითმეტიკის ოპერატორების მუშაობის
//      დემონსტრირება წილადი რიცხვებისათვის
double wiladi1, wiladi2, jami, sxvaoba, namravli, ganayofi, nashti;
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);

wiladi1 = Convert::ToDouble(textBox3->Text);
wiladi2 = Convert::ToDouble(textBox4->Text);
jami      = wiladi1 + wiladi2;
sxvaoba  = wiladi1 - wiladi2;
namravli = wiladi1 * wiladi2;
ganayofi = wiladi1 / wiladi2;
nashti   = ricxvi1 % ricxvi2;
label1->Text = jami.ToString();
```

```
label2->Text = sxvaoba.ToString();
label3->Text = namravli.ToString();
label4->Text = ganayofi.ToString();
label5->Text = nashti.ToString();
}
```

ინკრემენტისა და დეკრემენტის ოპერატორები

ინკრემენტის ოპერატორი (++) ერთით ზრდის თავისი ოპერანდის (არგუმენტის) მნიშვნელობას. ოპერანდი არის ცვლადი, რომელზეც უნდა შესრულდეს ინკრემენტის ოპერაცია. დეკრემენტის ოპერატორი (--) ერთით ამცირებს თავისი ოპერანდის მნიშვნელობას. მაგალითად,

```
x = x + 1;
```

ოპერატორი ასრულებს იმავე მოქმედებებს, რასაც

```
x++; ან ++x;
```

ოპერატორი. ანალოგიურად,

```
x = x - 1;
```

ოპერატორი ასრულებს იმავე მოქმედებებს, რასაც

```
x--; ან --x;
```

ოპერატორი.

როგორც ვხედავთ ეს ოპერატორები შეიძლება მიეთითოს როგორც ოპერანდამდე (**პრეფიქსი**), ისე ოპერანდის შემდეგ (**პოსტფიქსი**). ახლა ვნახოთ რა განსხვავებაა ამ პოზიციებს შორის. თუ ინკრემენტის ან დეკრემენტის ოპერატორი ოპერანდის წინაა, მაშინ ჯერ გაიზრდება ან შემცირდება ოპერანდის მნიშვნელობა, ხოლო შემდეგ მოხდება მისი გამოყენება გამოსახულებაში. თუ ინკრემენტის ან დეკრემენტის ოპერატორი მითითებულია ოპერანდის შემდეგ, მაშინ ჯერ მოხდება ამ ოპერანდის გამოყენება გამოსახულებაში და შემდეგ მისი მნიშვნელობის გაზრდა ან შემცირება.

ქვემოთ მოცემული პროგრამით ხდება ინკრემენტის ოპერატორის გამოყენების დემონსტრირება.

```
{
//      ++ ოპერატორის მუშაობის დემონსტრირება
int incr, shedegi;

incr = 5;
//      პრეფიქსური ინკრემენტი
shedegi = ++incr;                                   //      incr    = 6, shedegi = 6
label1->Text = incr.ToString();
label2->Text = shedegi.ToString();
incr = 5;
//      პოსტფიქსური ინკრემენტი
shedegi = incr++;                                   //      shedegi = 5, incr    = 6
label3->Text = incr.ToString();
label4->Text = shedegi.ToString();
}
```

თავდაპირველად, incr ცვლადს ენიჭება მნიშვნელობა 5. მომდევნო სტრიქონში ამ ცვლადის მარცხნივ წერია ++, ამიტომ მისი მნიშვნელობა ჯერ გაიზრდება ერთით და გახდება 6-ის ტოლი, შემდეგ კი მიენიჭება shedegi ცვლადს. შემდეგ, incr ცვლადს კვლავ ენიჭება

მნიშვნელობა 5. მომდევნო სტრიქონში ++ მოთავსებულია incr ცვლადის მარჯვნივ. ამიტომ, მისი მნიშვნელობა ჯერ მიენიჭება shedegi ცვლადს, შემდეგ კი გაიზრდება ერთით და გახდება 6-ის ტოლი.

ქვემოთ მოცემული პროგრამით ხდება დეკრემენტის ოპერატორის გამოყენების დემონსტრირება.

```
{
//      -- ოპერატორის მუშაობის დემონსტრირება
int decr, shedegi;

decr = 5;
//      პრეფიქსური დეკრემენტი
shedegi = --decr;                                   //      decr    = 4, shedegi = 4
label1->Text = decr.ToString();
label2->Text = shedegi.ToString();
decr = 5;
//      პოსტფიქსური დეკრემენტი
shedegi = decr--;                                   //      shedegi = 5, decr    = 4
label3->Text = decr.ToString();
label4->Text = shedegi.ToString();
}
```

შედარებისა და ლოგიკის ოპერატორები

შედარების ოპერატორი (2.7 ცხრილი) ადარებს ორ მნიშვნელობას და გასცემს bool ტიპის true ან false მნიშვნელობას. ლოგიკის ოპერატორი (2.8 ცხრილი) ასრულებს ლოგიკის ოპერაციას bool ტიპის მნიშვნელობებზე.

C++ ენაში == და != ოპერატორები შეიძლება გამოვიყენოთ ყველა ობიექტის მიმართ მათი შედარებისათვის ტოლობაზე ან უტოლობაზე. შედარების დანარჩენი ოპერატორები შეგვიძლია გამოვიყენოთ მხოლოდ მონაცემების ჩამოთვლადი ტიპების მიმართ, რომლებიც მოწესრიგებულია თავიანთ სტრუქტურაში. ასეთია მაგალითად, რიცხვების მოწესრიგებული სტრუქტურა 1, 2, 3 და ა.შ., ან ანბანის მიხედვით დალაგებული სიმბოლოები. შედეგად, შედარების ყველა ოპერატორი შეგვიძლია გამოვიყენოთ მონაცემთა ყველა რიცხვითი ტიპის მიმართ. bool ტიპის მნიშვნელობების შედარება შეიძლება მხოლოდ ტოლობაზე ან უტოლობაზე.

2.9 ცხრილში მოცემულია ლოგიკის ოპერატორების ჭეშმარიტების ცხრილი. როგორც ცხრილიდან ჩანს & ოპერატორი გასცემს true მნიშვნელობას მხოლოდ მაშინ, როდესაც მისი ორივე ოპერანდის მნიშვნელობაა true. წინააღმდეგ შემთხვევაში, ის გასცემს false მნიშვნელობას. | ოპერატორი გასცემს true მნიშვნელობას, თუ მისი ერთ-ერთი ოპერანდის მნიშვნელობაა true. წინააღმდეგ შემთხვევაში, ის გასცემს false მნიშვნელობას. ! ოპერატორი გასცემს თავისი ოპერანდის ლოგიკურად საწინააღმდეგო მნიშვნელობას. ^ ოპერატორი გასცემს false მნიშვნელობას თუ მის ორივე ოპერანდს ერთნაირი მნიშვნელობა აქვს, წინააღმდეგ შემთხვევაში, გასცემს true მნიშვნელობას.

ცხრილი 2.7. შედარების ოპერატორები

ოპერატორი	მნიშვნელობა
==	ტოლია
!=	არ არის ტოლი
>	მეტია
<	ნაკლებია
>=	მეტია ან ტოლი
<=	ნაკლებია ან ტოლი

ცხრილი 2.8. ლოგიკის ოპერატორები

ოპერატორი	მნიშვნელობა
&	AND (და)
|	OR (ან)
^	XOR (გამომრიცხავი ან)
&&	Short-circuit AND (სწრაფი **და** ოპერატორი)
||	Short-circuit OR (სწრაფი **ან** ოპერატორი)
!	NOT (არა)

ცხრილი 2.9. ლოგიკის ოპერატორების ჭეშმარიტების ცხრილი

B1	B2	B1 & B2	B1 | B2	B1 ^ B2	!B1
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

ამ პროგრამით ხდება ლოგიკის ოპერატორების მუშაობის დემონსტრირება.

```
{
//      ლოგიკის ოპერატორების მუშაობის დემონსტრირება
bool b1, b2, b3, b4, b5, b6;

b1 = true;
b2 = false;
b3 = b1 & b2;                   //      b3 = false
b4 = b1 | b2;                   //      b4 = true
b5 = b1 ^ b2;                   //      b5 = true
b6 = !b1;                       //      b6 = false
label1->Text = b3.ToString();
label2->Text = b4.ToString();
label3->Text = b5.ToString();
label4->Text = b6.ToString();
}
```

მოცემული პროგრამით ხდება შედარების ოპერატორების მუშაობის დემონსტრირება.

```
{
//      შედარების ოპერატორების მუშაობის დემონსტრირება
int ricxvi1 = 5, ricxvi2 = 10;
bool shedegi;
```

```
shedegi = ricxvi1 > 0;                          //      shedegi = true
label3->Text = shedegi.ToString();
shedegi = ( ricxvi1 < 2 ) && ( ricxvi2 > 7 );   //      shedegi = false
label1->Text = shedegi.ToString();
shedegi = ( ricxvi1 == 3 ) || ( ricxvi2 != 5 ); //      shedegi = true
label2->Text = shedegi.ToString();
}
```

&& და || ლოგიკის სწრაფი ოპერატორებია. მათი გამოყენებით პროგრამა უფრო სწრაფად სრულდება. თუ && ოპერატორის შესრულებისას პირველმა ოპერანდმა მიიღო false მნიშვნელობა, მაშინ შედეგი იქნება false მიუხედავად მეორე ოპერანდის მნიშვნელობისა. თუ || ოპერატორის შესრულებისას პირველმა ოპერანდმა მიიღო true მნიშვნელობა, მაშინ შედეგი იქნება true მიუხედავად მეორე ოპერატორის მნიშვნელობისა. ასეთ შემთხვევებში აღარ არის საჭირო მეორე ოპერანდის შეფასება, რადგან მისი მნიშვნელობა საბოლოო შედეგს ვერ შეცვლის. შედეგად, პროგრამა უფრო სწრაფად სრულდება.

მინიჭების ოპერატორი

მისი სინტაქსია:

ცვლადი = გამოსახულება;

ცვლადს და გამოსახულებას ერთნაირი ტიპი უნდა ჰქონდეს. მინიჭების ოპერატორი ცვლადს ანიჭებს გამოსახულების მნიშვნელობას. შეგახსენებთ, რომ მინიჭების ოპერატორის მარჯვნივ მოთავსებულ გამოსახულებას უნდა ჰქონდეს მინიჭების ოპერატორის მარცხნივ მოთავსებული ცვლადის ტიპი.

მინიჭების ოპერატორი საშუალებას იძლევა, აგრეთვე, შევქმნათ მინიჭებების მიმდევრობა:

```
int ricxvi1, ricxvi2, ricxvi3;
ricxvi1 = ricxvi2 = ricxvi3 = 50;
```

აქ სამივე ცვლადს ენიჭება მნიშვნელობა 50. ასეთი მინიჭება საშუალებას გვაძლევს რამდენიმე ცვლადს ერთდროულად მივანიჭოთ ერთი და იგივე მნიშვნელობა.

მინიჭების შედგენილი ოპერატორი

მინიჭების შედგენილ (შემოკლებულ) ოპერატორში არითმეტიკის ოპერატორები მოთავსებულია მინიჭების ოპერატორის მარცხნივ. მისი სინტაქსია:

ცვლადი ოპერატორი= გამოსახულება;

სადაც, ***ოპერატორი*** არის არითმეტიკის ან ლოგიკის ოპერატორი. მაგალითად, გამოსახულება:

```
jami = jami + 25;
```

შედგენილი ოპერატორის გამოყენებით შეგვიძლია ასე ჩავწეროთ:

```
jami += 25;
```

+= ოპერატორების წყვილი მიუთითებს, რომ jami ცვლადს უნდა მიენიჭოს მნიშვნელობა jami + 25.

შედგენილი (შემოკლებული) ოპერატორებია:

```
+=      -=      *=      /=      %=      &=      |=      ^=
```

მინიჭების შედგენილი ოპერატორები მინიჭების ჩვეულებრივ ოპერატორზე კომპაქტურია. ეს, განსაკუთრებით იგრძნობა მაშინ, როდესაც გრძელ სახელებს ვიყენებთ. ამ

შემთხვევაში, სახელის შეტანა ერთხელ გვიწევს. მეორეც, მათი გამოყენება აჩქარებს კოდის კომპილაციას, რადგან ოპერანდის შეფასება მხოლოდ ერთხელ ხდება.

ოპერატორის პრიორიტეტი

ჩვეულებრივ, გამოსახულებაში ჯერ სრულდება პრიორიტეტული ოპერაციები, შემდეგ კი - ნაკლებად პრიორიტეტული. 2.10 ცხრილში ოპერაციები დალაგებულია პრიორიტეტების კლების მიხედვით. შესაბამისად, ცხრილის ზედა ნაწილში მოთავსებულია მაღალი პრიორიტეტის ოპერაციები, ქვედა ნაწილში კი - დაბალი პრიორიტეტის. ერთ სტრიქონში მოთავსებულ ოპერაციებს ერთნაირი პრიორიტეტი აქვს. თუ გამოსახულებაში გამოყენებული არ არის ფრჩხილები, მაშინ თანაბარი პრიორიტეტის მქონე ოპერაციები შესრულდება იმ მიმდევრობით, რომელსაც განსაზღვრავს მათი **ასოციაციურობა**. „მარცხენა“ ასოციაციურობის შემთხვევაში გამოსახულების ყველაზე მარცხენა ოპერაცია შესრულდება პირველ რიგში, შემდეგ კი მიმდევრობით სრულდება ყველა ოპერაცია მარცხნიდან მარჯვნივ. მაგალითად,

```
shedegi = ricxvi1 + ricxvi2 + ricxvi3;
```

გამოსახულებაში შეკრების ოპერაციები შესრულდება იმ მიმდევრობით, რა მიმდევრობითაც არის ჩაწერილი, რადგანაც + ოპერაციას აქვს მარცხენა ასოციაციურობა. ანუ ჯერ შეიკრიბება ricxvi1 და ricxvi2. მიღებულ ჯამს დაემატება ricxvi3.

როგორც ცხრილიდან ჩანს, *, / და % ოპერატორებს უფრო მაღალი პრიორიტეტი აქვთ, ვიდრე + და - . განვიხილოთ გამოსახულება:

```
y = x1 + x2 * x3 - x4 / x5 ;
```

ის შემდეგნაირად გამოითვლება. გამოსახულებაში პირველია + ოპერატორი. სანამ ეს ოპერატორი შესრულდება კომპილატორი ამოწმებს მის მარჯვნივ რომელი ოპერატორია. ესაა *, რომელსაც უფრო მაღალი პრიორიტეტი აქვს, ვიდრე + ოპერატორს. შემდეგ, კომპილატორი ამოწმებს * ოპერატორის მარჯვნივ მდებარე ოპერატორს. ესაა -, რომელსაც უფრო დაბალი პრიორიტეტი აქვს, ვიდრე * ოპერატორს. ამიტომ პირველ რიგში შესრულდება * ოპერატორი. რადგან + და - ოპერატორებს თანაბარი პრიორიტეტები აქვს და + რიგით პირველია გამოსახულებაში, ამიტომ შემდეგ შესრულდება + ოპერატორი. სანამ კომპილატორი შეასრულებს - ოპერატორს, ის ამოწმებს მის მარჯვნივ რომელი ოპერატორია მოთავსებული. ესაა / ოპერატორი, რომელსაც უფრო მაღალი პრიორიტეტი აქვს ვიდრე - ოპერატორს. ამიტომ, ჯერ შესრულდება / ოპერატორი, შემდეგ კი - ოპერატორი.

ახლა განვიხილოთ მეორე გამოსახულება:

```
y = ( x1 + x2 ) * x3 - ( x4 + x5 ) / x6 ;
```

აქ ჯერ შესრულდება პირველ მრგვალ ფრჩხილებში მოთავსებული გამოსახულება. შემდეგ, * ოპერატორი. შემდეგ, მეორე მრგვალ ფრჩხილებში მოთავსებული გამოსახულება. შემდეგ, / ოპერატორი და ბოლოს, - ოპერატორი.

როგორც ვხედავთ, პრიორიტეტების ცოდნა საჭიროა იმისათვის, რომ სწორად ჩავწეროთ გამოსახულება და შესაბამისად, მივიღოთ სწორი შედეგი.

უნარულია ოპერაცია, რომელსაც ერთი ოპერანდი აქვს. **ბინარულია** ოპერაცია, რომელსაც ორი ოპერანდი აქვს. ***ტერნარულია*** ოპერაცია, რომელსაც სამი ოპერანდი აქვს. უნარულ ოპერაციას ყოველთვის უფრო მაღალი პრიორიტეტი აქვს ვიდრე - ბინარულს.

გამოსახულება. მათემატიკის ფუნქციები

გამოსახულება არის ოპერატორების, ცვლადებისა და ფუნქციების კომბინაცია.

მაგალითად,

```
y = x + z - 5;
y = x - sin(x) + 10.97;
```

პროგრამის კითხვა რომ გაუმჯობესდეს გამოსახულებებში ტაბულირების სიმბოლოები და ინტერვალები გამოიყენება. მაგალითად, მოცებული ორი გამოსახულებიდან მეორის წაკითხვა უფრო ადვილია, ვიდრე პირველის:

```
y=x/5+z*(w-2);
y = x / 5 + z * ( w - 2 );
```

მრგვალი ფრჩხილები ზრდის მასში მოთავსებული ოპერანდების პრიორიტეტს. ისინი აქ ისევე გამოიყენება, როგორც ალგებრაში. მრგვალი ფრჩხილები გამოიყენება, აგრეთვე, პროგრამის კოთხვის გასაუმჯობესებლად. მაგალითად, მოცემული ორი სტრიქონიდან მეორე უკეთესად იკითხება:

```
y = x * 5 + z / 7.2 - w;
y = ( x * 5)  + ( z / 7.2 ) - w;
```

მათემატიკის ფუნქციები აღწერილია math.h სტანდარტულ ბიბლიოთეკაში (2.11 ცხრილი). მათი უმრავლესობა double ტიპს იყენებს. არსებობს ამ ფუნქციებთან მუშაობის ორი გზა:

1. მათ გამოსაძახებლად ჯერ ვწერთ კლასის სახელს Math, შემდეგ ორ წერტილს და ფუნქციის სახელს. მაგალითად, Math::Ceiling(x).
2. ფაილის დასაწყისში ვათავსებთ #include <cmath> დირექტივას და Math სახელის მითითება აღარ მოგვიწევს.

მოკლედ განვიხილოთ ზოგიერთი ფუნქცია. Pow(x, y) ფუნქციას x რიცხვი აჰყავს y ხარისხში. მაგალითად, Pow(5, 2) ფუნქციის შესრულების შედეგად გაიცემა 25. Exp(x) ფუნქციას e რიცხვი აჰყავს x ხარისხში. მაგალითად, Exp(1) ფუნქციის შესრულების შედეგად გაიცემა 2.71 . Max(x, y) ფუნქცია გასცემს x და y რიცხვებს შორის უდიდესს, Min(x, y) ფუნქცია კი - უმცირესს. Floor(x) ფუნქცია x რიცხვს ამრგვალებს ნაკლებობით. მაგალითად, Floor(5.998) ფუნქციის შესრულების შედეგად გაიცემა 5. Ceiling(x) ფუნქცია x რიცხვს ამრგვალებს მეტობით. მაგალითად, Ceiling(5.123) ფუნქციის შესრულების შედეგად გაიცემა 6. Round(x, y) ფუნქცია გამოიყენება x რიცხვის დაფორმატებული გამოტანისათვის. y მიუთითებს ათწილადის თანრიგების რაოდენობას. მაგალითად, Round(5.1234, 2) ფუნქციის შესრულების შედეგად გაიცემა ათწილადი 5.12. Sign(x) ფუნქცია გასცემს x რიცხვის ნიშანს. მაგალითად, Sign(-5) ფუნქციის შესრულების შედეგად გაიცემა -1, ხოლო Sign(5) ფუნქციის შესრულების შედეგად კი - 1.

შევადგინოთ პროგრამა, რომლის შესრულებითაც მოხდება კვადრატული ფესვის ამოღებას რიცხვიდან, რომელიც textBox1 კომპონენტში შეგვაქვს.

```
{
//        კვადრატული ფესვის ამოღება
double ricxvi, shedegi;

ricxvi    = Convert::ToDouble(textBox1->Text);
shedegi = Math::Sqrt(ricxvi);
label1->Text = shedegi.ToString();
label2->Text = Math::Round(shedegi,2).ToString();                //        შედეგი დამრგვალდება
}
```

შევადგინოთ კიდევ ერთი პროგრამა, რომელიც გამოთვლის მოცემული გამოსახულების მნიშვნელობას:

$$y = \frac{\sqrt{x^5 + \sin x}}{1 - e^x}$$

ცხრილი 2.10. ოპერაციების პრიორიტეტები

ოპერაცია	ასოციაციურობა
::	მარცხენა
( ) [ ] ->	მარცხენა
! “ + (უნარული) -(უნარული) ++ -- &(უნარული) *(უნარული) (ტიპის გარდაქმნა) static_cast const_cast dynamic_cast reinterpret_cast sizeof new delete . . . typeid	მარჯვენა
. *(უნარული) ->*	მარცხენა
* / %	მარცხენა
+ -	მარცხენა
<< >>	მარცხენა
< <= > >=	მარცხენა
== !=	მარცხენა
&	მარცხენა
^	მარცხენა
|	მარცხენა
&&	მარცხენა
||	მარცხენა
?:(პირობის ოპერაცია)	მარჯვენა
= *= /= %= += -= &= ^= |= <<= >>=	მარჯვენა
‘	მარცხენა

ცხრილი 2.11. მათემატიკის ფუნქციები

ფუნქცია	დანიშნულება
Abs(x)	აბსოლუტური მნიშვნელობა
Sin(x)	სინუსი
Cos(x)	კოსინუსი
Tan(x)	ტანგენსი
Sqrt(x)	კვადრატული ფესვი
Exp(x)	ექსპონენტა
Log(x)	ლოგარითმი
Log10(x)	ათობითი ლოგარითმი
Max(x, y)	ორ რიცხვს შორის უდიდესი
Min(x, y)	ორ რიცხვს შორის უმცირესი
Pow(x, y)	ახარისხება
Floor(x)	დამრგვალება ნაკლებობით
Ceiling(x)	დამრგვალება მეტობით
Round(x, y)	ფორმატირებული გამოტანა
E	2,71
PI	3,14
Sign(x)	არგუმენტის ნიშანი

```
{
//        ფორმულის გამოთვლა
double ricxvi, shedegi;

ricxvi  = Convert::ToDouble(textBox1->Text);
shedegi =  Math::Sqrt(Math::Pow(ricxvi,5) + Math::Sin(ricxvi) ) / ( 1 - Math::Exp(ricxvi) );
label1->Text = shedegi.ToString();
}
```

1-ელ დანართში მოცემულია ამოცანები თავების მიხედვით.

ტიპის გარდაქმნა

გამოთვლები შეიძლება შესრულდეს მხოლოდ ერთი ტიპის მნიშვნელობებს შორის. როდესაც გამოსახულება შეიცავს სხვადასხვა ტიპის ცვლადებსა და კონსტანტებს (მუდმივებს), მაშინ თითოეული ოპერაციის შესრულების წინ კომპილატორს მოუწევს ტიპების გარდაქმნის შესრულება. ტიპების გარდაქმნის პროცესს ***დაყვანა*** ეწოდება. მაგალითად, თუ გვინდა შევკრიბოთ int და double ტიპის რიცხვები, მაშინ int ტიპის რიცხვი ჯერ გარდაიქმნება double ტიპად და შემდეგ შესრულდება შეკრება. int ტიპის რიცხვის მნიშვნელობა არ იცვლება. უბრალოდ წილადად გარდაქმნილი მნიშვნელობა მეხსიერებაში ინახება გამოთვლების დამთავრებამდე.

გამოსათვლელი გამოსახულება იყოფა ოროპერანდიან რამდენიმე ოპერაციად. მაგალითად,

```
shedegi = 8 / 2 + 9 - 3;
```

გამოსახულება შემდეგი ოროპერანდიანი ოპერაციებისაგან შედგება:

1) 8 / 2. შედეგად მიიღება 4;
2) შემდეგ, შესრულდება 4 + 9 ოპერაცია. შედეგად მიიღება 13;
3) შემდეგ, შესრულდება 13 – 3 ოპერაცია. შედეგად მიიღება 10.

როგორც ვხედავთ, ოპერანდების დაყვანის ოპერაცია საჭიროა მხოლოდ ოპერანდების წყვილების მიმართ გადაწყვეტილების მიღების დროს. ამიტომ, სხვადასხვა ტიპის ოპერანდის წყვილისთვის მოწმდება ქვემოთ მოცემული წესები იმ მიმდევრობით, როგორც არიან ისინი მოცემული. თუ წესის გამოყენება შეიძლება ოპერანდების კონკრეტული წყვილის მიმართ, მაშინ მოხდება ამ წესის გამოყენება. ოპერანდების დაყვანის ეს წესებია:

1. თუ ერთი ოპერანდის ტიპია long double, მაშინ მეორე ოპერანდიც გარდაიქმნება ამ ტიპის ოპერანდად.
2. თუ ერთი ოპერანდის ტიპია double, მაშინ მეორე ოპერანდიც გარდაიქმნება ამ ტიპის ოპერანდად.
3. თუ ერთი ოპერანდის ტიპია float, მაშინ მეორე ოპერანდიც გარდაიქმნება ამ ტიპის ოპერანდად.
4. char, signed char, unsigned char, short ან unsigned short ტიპის ოპერანდი გარდაიქმნება int ტიპის ოპერანდად.
5. ჩამოთვლადი ტიპი გარდაიქმნება int, unsigned int, long ან unsigned long ტიპად, იმაზე დამოკიდებულებით, თუ რომელი ტიპია საკმარისი, რომ დაიტიოს ჩამოთვლების დიაპაზონი.
6. თუ ერთი ოპერანდის ტიპია unsigned long, მაშინ მეორე ოპერანდიც გარდაიქმნება ამ ტიპის ოპერანდად.
7. თუ ერთი ოპერანდის ტიპია long, მეორე ოპერანდი ტიპი კი - unsigned int, მაშინ ორივე

ოპერადი გარდაიქმნება unsigned long ტიპის ოპერანდად.

8. თუ ერთი ოპერანდის ტიპია long, მაშინ მეორე ოპერანდიც გარდაიქმნება ამ ტიპის ოპერანდად.

ტიპების გარდაქმნის საბაზო პრინციპი ასეთია: ყოველთვის გარდაიქმნება ის ოპერანდი, რომლის ტიპის დიაპაზონი ნაკლებია მეორე ოპრანდის ტიპის დიაპაზონზე. მაგალითად, თუ პირველი ოპერანდის ტიპია int და მეორე ოპერანდის ტიპი - long, მაშინ პირველი ოპერანდის ტიპი გარდაიქმნება long ტიპად. ეს, თავის მხრივ, ზრდის სწორი შედეგის მიღების ალბათობას. მაგალითი:

```
{
double wiladi_d = 25.0;
int mteli = 10;
float wiladi_f = 4.0f;
char simbolo = 5;

wiladi_d = ( wiladi_d + mteli ) / ( mteli - simbolo ) * wiladi_f - simbolo * wiladi_f;
label1->Text = wiladi_d.ToString();
}
```

პროგრამაში გამოითვლება შემდეგი გამოსახულების მნიშვნელობა:

```
wiladi_d = ( wiladi_d + mteli ) / ( mteli - simbolo ) * wiladi_f - simbolo * wiladi_f;
```

პირველად გამოითვლება (wiladi_d + mteli) გამოსახულების მნიშვნელობა. + ოპერაციის ოპერანდებს სხვადასხვა ტიპი აქვს, კერძოდ კი double და int, ამიტომ მისი გამოთვლისას გამოყენებული იქნება მეორე წესი. ამ წესის თანახმად mteli ცვლადის ტიპი გარდაიქმნება double ტიპად. ამ გარდაქმნის შემდეგ შესრულდება შეკრების ოპერაცია. შედეგი იქნება 35.0 .

შემდეგ გამოითვლება (mteli - simbolo) გამოსახულების მნიშვნელობა. - ოპერაციის ოპერანდებს აქვს int და char ტიპი, ამიტომ გამოყენებული იქნება მეოთხე წესი. ამ წესის თანახმად simbolo ცვლადის ტიპი გარდაიქმნება int ტიპად. ამ გარდაქმნის შემდეგ შესრულდება გამოკლების ოპერაცია. შედეგი იქნება 5.

შემდეგ, 35.0 გაიყოფა 5-ზე. მეორე წესის თანახმად 5 გარდაიქმნება წილადად (5.0) და შესრულდება გაყოფის ოპერაცია. შედეგი იქნება 7.0. შემდეგ, ეს წილადი უნდა გამრავლდეს 4.0-ზე. ამ შემთხვევაშიც, ვიყენებთ მეორე წესს, რომლის თანახმად 4.0 უნდა გარდაიქმნას double ტიპის წილადად. გარდაქმნის შემდეგ შესრულდება გამრავლების ოპერაცია. შედეგი იქნება 28.0.

შემდეგ, გამოითვლება simbolo * wiladi_f გამოსახულება. აქ გამოყენებული იქნება მესამე წესი. შედეგად, simbolo ოპერანდის ტიპი გარდაიქმნება float ტიპად. გამოთვლის შედეგი იქნება 20.0.

ბოლოს შესრულდება გამოკლების ოპერაცია. რადგან 28.0 რიცხვის ტიპია double, ამიტომ გამოყენებული იქნება მეორე წესი. რიცხვი 20.0 float ტიპიდან გარდაიქმნება double ტიპად. გამოკლების ოპერაციის შედეგი იქნება 8.0. ეს მნიშვნელობა მიენიჭება wiladi_d ცვლადს.

დაყვანა მინიჭების ოპერატორებში

ჩვენ შეგვიძლია გამოვიწვიოთ არაცხადი გარდაქმნა, თუ მინიჭების ოპერატორის მარჯვნივ მოვათავსებთ გამოსახულებას, რომლის ტიპი განსხვავდება მინიჭების ოპერატორის მარცხნივ მოთავსებული ცვლადის ტიპისაგან. ამან შეიძლება გამოიწვიოს მნიშვნელობის შეცვლა და, შესაბამისად, მონაცემების დაკარგვა. მაგალითი:

```
{
//          პროგრამით ხდება არაცხადი გარდაქმნის დემონსტრირება
int mteli = 0;
```

```
float wiladi = 3.7f;
mteli = wiladi;                                    //        mteli ცვლადს მიენიჭება 3
label1->Text = mteli.ToString();
}
```

აქ mteli მთელი ტიპის ცვლადს ენიჭება wiladi წილადი ცვლადის მნიშვნელობა. ამ მინიჭების დროს წილადი ნაწილი იკარგება (0.7) და mteli ცვლადს ენიჭება მნიშვნელობა 3.

ცხადი დაყვანა

თუ გამოსახულება შეიცავს სხვადასხვა ტიპის ცვლადებს, მაშინ კომპილატორი, საჭიროების შემთხვევაში, ავტომატურად ასრულებს ტიპების დაყვანას. არსებობს ტიპების დაყვანის ოთხი სახე:

1. **static_cast** - სტატიკური დაყვანა. ტიპის დაყვანა სრულდება საწყისი კოდის კომპილირებისას. პროგრამის შესრულებისას დაყვანის უსაფრთხოების შემოწმება აღარ შესრულდება.
2. **dynamic_cast** - დინამიკური დაყვანა. ტიპის დაყვანის უსაფრთხოება შემოწმდება პროგრამის შესრულებისას.
3. **const_cast** - გამორიცხავს გამოსახულების მუდმივობას.
4. **reinterpret_cast** - უპირობო დაყვანა.

ჩვენ შეგვიძლია იძულებით მივუთითოთ ერთი ტიპის მეორე ტიპზე დაყვანა, რასაც ***ცხადი დაყვანა*** ეწოდება. მისი სინტაქსია:

static_cast <*დაყვანის_ტიპი*> (*გამოსახულება*)

აქ **static_cast** საკვანძო სიტყვა მიუთითებს, რომ სრულდება ***სტატიკური დაყვანა***. ***გამოსახულების*** მნიშვნელობა დაყვანილი უნდა იყოს ***დაყვანის_ტიპზე***. მაგალითი:

```
{
//        პროგრამით ხდება ცხადი დაყვანის დემონსტრირება
double wiladi1 = 45.8;
double wiladi2 = 20.3;
int mteli;

mteli = static_cast <int> (wiladi1) - static_cast <int> (wiladi2);          //        mteli = 25
label1->Text = mteli.ToString();
}
```

პრორამით გამოითვლება შემდეგი გამოსახულების მნიშვნელობა:

```
mteli = static_cast <int> (wiladi1) - static_cast <int> (wiladi2);
```

გამოკლების შესრულებამდე სრულდება ტიპების დაყვანა. wiladi1 ცვლადი გარდაიქმნება მთელ რიცხვად და მისი მნიშვნელობა გახდება 45. ადგილი აქვს მნიშვნელობის დაკარგვას. იგივე ეხება wiladi2 ცვლადს. მისი მნიშვნელობა გახდება 20-ის ტოლი. გამოკლების შედეგად მიიღება მთელი რიცხვი 20, რომელიც მიენიჭება mteli ცვლადს.

ტიპების დაყვანისას, უნდა გვახსოვდეს, რომ შეიძლება დაიკარგოს ინფორმაცია. მაგალითი:

```
{
//        ინფორმაციის დაკარგვის დემონსტრირება
double wiladi1 = 0.8;
float    wiladi2 = 0.7f;
long mteli1;
int   mteli2;
```

```
mteli1 = wiladi1;                    //      mteli1 = 0
mteli2 = wiladi2;                    //      mteli2 = 0
label1->Text = mteli1.ToString() + "   " + mteli2.ToString();
}
```

ამ პროგრამაში,

```
mteli1 = wiladi1;
mteli2 = wiladi2;
```

მინიჭებების შედეგად mteli1 და mteli2 ცვლადებს მიენიჭებათ 0-ები და შესაბამისად, წილადი ნაწილები იკარგება.

double ტიპის დაყვანამ float ტიპზე შეიძლება გამოიწვიოს ინფორმაციის დაკარგვა. ანალოგიურად, long ტიპის დაყვანამ int ტიპზე შეიძლება გამოიწვიოს ინფორმაციის დაკარგვა და ა.შ. ამიტომ, უმჯობესია ტიპების დაყვანა გამოვიყენოთ მხოლოდ აუცილებლობის შემთხვევაში.

safe_cast ოპერაცია

არსებობს ტიპების დაყვანის კიდევ ერთი უსაფრთხო საშუალება. safe_cast ოპერაცია გამოიყენება ტიპების აშკარა უპრობლემო გარდაქმნისათვის CLR გარემოში. მაგალითი:

```
{
//      ტიპების უსაფრთხო დაყვანა
double wiladi1 = Convert::ToDouble(textBox1->Text);
double wiladi2 = Convert::ToDouble(textBox2->Text);

int mteli = safe_cast<int>(wiladi1) + safe_cast<int>(wiladi2);
label1->Text = mteli.ToString();
}
```

ტიპების დაყვანის ძველი სტილი

ტიპების დაყვანის ძველი სტილის სინტაქსია:

(დაყვანის_ტიპი) გამოსახულება;

მაგალითი:

```
{
//      ტიპების დაყვანის ძველი სტილის დემონსტრირება
double wiladi1 = 45.8;
double wiladi2 = 20.3;
int mteli;
mteli = (int) wiladi1 - (int) wiladi2;          //      mteli = 25
label1->Text = mteli.ToString();
}
```

ტიპების დაყვანის ძველი სტილის ნაკლია ის, რომ იგი ნაკლებად მდგრადია შეცდომების მიმართ, ამიტომ მისი გამოყენება CLR პროგრამებში რეკომენდებული არ არის.

შენახვის დრო და ხილვადობის უბანი

პროგრამის შესრულებისას ყველა ცვლადს აქვს სიცოცხლის (არსებობის) შეზღუდული

დრო. ისინი არსებობას იწყებენ მათი გამოცხადების მომენტიდან და არსებობენ (ცოცხლობენ) გარკვეულ მომენტამდე, მაგრამ არაუგვიანეს პროგრამის დასრულების მომენტისა. ცვლადის არსებობის პერიოდი განისაზღვრება ***შენახვის დროით*** (storage duration). არსებობს ცვლადების შენახვის სამი სახე:

- ავტომატური;
- სტატიკური;
- დინამიკური.

იმაზე დამოკიდებულებით თუ როგორ ვქმნით ცვლადს, ის შეიძლება იყოს ავტომატური, სტატიკური ან დინამიკური. გარდა ამისა, ცვლადებს ახასიათებს ხილვადობის უბანი (scope). ***ცვლადის ხილვადობის უბანი*** არის პროგრამის ნაწილი, რომელშიც ცვლადის სახელი განსაზღვრულია. ხილვადობის უბნის გარეთ ჩვენ არ შეგვიძლია მივმართოთ ცვლადს, თუმცა ის შეიძლება არსებობდეს ამ უბნის გარეთ.

ავტომატური ცვლადი

ცვლადს, რომელსაც ვაცხადებდით ბლოკში ანუ ფიგურულ ფრჩხილებში ***ავტომატური*** ეწოდება. ბლოკი არის ოპერატორების ერთობლიობა. ის შეიძლება შედგებოდეს ერთი ან მეტი ოპერატორისაგან. ბლოკი იწყება "{" სიმბოლოთი და მთავრდება "}" სიმბოლოთი. ავტომატურ ცვლადს აქვს ***ლოკალური ხილვადობის უბანი*** ანუ ***ბლოკის ხილვადობის უბანი***. ავტომატური ცვლადი ხილულია დაწყებული გამოცხადების ადგილიდან (წერტილიდან) მისი გამოცხადების შემცველი ბლოკის ბოლომდე. მეხსიერების უბანი, რომელსაც ავტომატური ცვლადი იკავებს, ავტომატურად გამოიყოფა სტეკში.

სტეკი არის სია (მეხსიერების უბანი), რომლის ელემენტებთან მიმართვა სრულდება პრინციპით - „უკანასკნელი მოვიდა - პირველი წავიდა“ (LIFO – last-in, first-out). მაგალითად, თუ სტეკში ჯერ ჩავწერთ 1-ს, შემდეგ 7-ს და ბოლოს 5-ს, მაშინ წაკითხვისას, ჯერ წაიკითხება 5, შემდეგ 7 და ბოლოს - 4. თითოეულ პროგრამას საკუთარი სტეკი გამოეყოფა.

ავტომატური ცვლადი იქმნება მისი გამოცხადების დროს და არსებობას ამთავრებს მისი გამოცხადების შემცველი ბლოკის ბოლოს ანუ იქ, სადაც იმყოფება დამხურავი ფიგურული ფრჩხილი. ყოველთვის, როდესაც სრულდება ოპერატორების ბლოკი, რომელიც შეიცავს ავტომატური ცვლადის გამოცხადებას, ავტომატური ცვლადი ხელახლა იქმნება და თუ განსაზღვრულია მისი საწყისი მნიშვნელობა, ის ხელახლა ინიცირდება ამ მნიშვნელობით ყოველი შექმნის დროს. როდესაც ავტომატური ცვლადი იშლება, მის მიერ დაკავებული მეხსიერება თავისუფლდება.

ავტომატური ცვლადის აღნიშვნისათვის შეგვიძლია **auto** საკვანძო სიტყვა გამოვიყენოთ. თუმცა, მისი მითითება აუცილებელი არ არის, რადგან ის ავტომატურად იგულისხმება. მაგალითი:

```
{
//      ავტომატურ ცვლადებთან მუშაობის დემონსტრირება
int ricxvi1 = 10;
int ricxvi2 = 20;
{
//      ხილვადობის ახალი უბნის დასაწყისი
int ricxvi1 = 30;
int ricxvi3 = 40;
ricxvi1 += 30;                  //      შიგა ricxvi1 = 60
ricxvi3 += 40;                  //      შიგა ricxvi3 = 80
label1->Text = L"შიგა ricxvi1 = " + ricxvi1.ToString() + L"\nშიგა ricxvi3 = " + ricxvi3.ToString();
```

```
//        ხილვადობის ახალი უბნის დასასრული
}
ricxvi1 += 10;                    //        გარე ricxvi1 = 20
ricxvi2 += 20;                    //        გარე ricxvi2 = 40
label2->Text = L"გარე ricxvi1 = " + ricxvi1.ToString() + L"\nგარე ricxvi2 =  " + ricxvi2.ToString();
}
```

როგორც ვხედავთ, ხილვადობის ახალ უბანში გამოცხადებულია ricxvi1 ცვლადი. ის განსხვავდება ადრე გამოცხადებული ricxvi1 ცვლადისაგან. ორივე ეს ცვლადი ოპერატიული მეხსიერების სხვადასხვა უბანშია მოთავსებული, ამიტომ სხვადასხვა ცვლადია და შესაბამისად, სხვადასხვა მნიშვნელობა აქვთ. შიგა ricxvi1 ცვლადი ფარავს გარე ricxvi1 ცვლადს. ამიტომ, ხილვადობის ახალ უბანში ჩვენ ვერ მივმართავთ გარე ricxvi1 ცვლადს. გარე ricxvi1 ცვლადი ხილული გახდება ხილვადობის უბნიდან გამოსვლისთანავე.

სტატიკური ცვლადი

სტატიკურია ცვლადი, რომელიც განსაზღვრულია როგორც ლოკალური, მაგრამ აგრძელებს არსებობას იმ ბლოკიდან გამოსვლის შემდეგ, რომელშიც ის არის გამოცხადებული. სტატიკური ცვლადის გამოცხადება ხდება static სპეციფიკატორის გამოყენებით. მაგალითი:

```
{
//        სტატიკურ ცვლადთან მუშობის დემონსტრირება
static int ricxvi1 = 5;
{
static int ricxvi2 = 10;
ricxvi2++;
label2->Text = ricxvi2.ToString();
}
ricxvi1++;
label1->Text = ricxvi1.ToString();
}
```

სინამდვილეში სტატიკური ცვლადი არსებობს პროგრამის შესრულების მთელი დროის განმავლობაში, თუნდაც ის გამოცხადებული იყოს ბლოკის შიგნით. სტატიკურ ცვლადს აქვს ბლოკის ხილვადობის უბანი და შენახვის სტატიკური დრო. თუ სტატიკურ ცვლადს არ მივანიჭებთ საწყის მნიშვნელობას, მაშინ მას გაჩუმებით 0 მიენიჭება.

გლობალური ცვლადი

ცვლადს, რომელიც გამოცხადებულია ყველა ბლოკისა და კლასის გარეთ, ***გლობალური*** ეწოდება. მათ აქვთ ***გლობალური ხილვადობის უბანი***, რომელსაც აგრეთვე, ***გლობალური სახელების სივრცის ხილვადობის უბანი*** ან ***ფაილის ხილვადობის უბანი*** ეწოდება. გლობალური ცვლადი მისაწვდომია ამ ფაილში განსაზღვრული ყველა ფუნქციიდან დაწყებული ამ ცვლადის განსაზღვრის ადგილიდან. თუ გლობალური ცვლადი განსაზღვრულია ფაილის დასაწყისში, მაშინ იგი მისაწვდომი იქნება ფაილის ნებისმიერი ადგილიდან.

გლობალურ ცვლადს ნაგულისხმევი აქვს ***სიცოცხლის სტატიკური დრო***. ამიტომ, იგი არსებობს პროგრამის დაწყების მომენტიდან მისი დამთავრების მომენტამდე. თუ გლობალურ ცვლადს არ აქვს მინიჭებული საწყისი მნიშვნელობა, მაშინ მას ნაგულისხმევად მიენიჭება ნული. გლობალური ცვლადის გამოცხადება შეგვიძლია ფაილის დასაწყისში #pragma once

დირექტივის წინ ან შემდეგ, აგრეთვე, #pragma endregion დირექტივის წინ ან შემდეგ. მაგალითი:

```
//          გლობალურ ცვლადებთან მუშობის დემონსტრირება
#pragma once
int ricxvi1 = 10;
static int ricxvi2 = 20;

namespace Prog1 {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
. . .
#pragma endregion
int ricxvi3;
static int ricxvi4 = 40;
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ricxvi1 = 1;
ricxvi2 = 2;
ricxvi3 = 3;
ricxvi4 = 4;
int shedegi = ricxvi1 + ricxvi2 + ricxvi3 + ricxvi4;                    //          shedegi = 10
label1->Text = shedegi.ToString();
}
```

აქ ricxvi1 და ricxvi2 გლობალური ცვლადებია. ისინი გამოცხადებულია #pragma once დირექტივის შემდეგ და ხდება მათი ინიციალიზება. გლობალურია, აგრეთვე ricxvi3 და ricxvi4 ცვლადები. ისინი გამოცხადებულია #pragma endregion დირექტივის შემდეგ. მათი ინიციალიზებისათვის, ისინი უნდა გამოვაცხადოთ როგორც სტატიკური.

შეგვიძლია პროგრამის ყველა ცვლადი გამოვაცხადოთ როგორც გლობალური. ასეთ შემთხვევაში გაიზრდება მათი არასწორად შეცვლის რისკი. როდესაც ცვლადი ავტომატურია, ის არსებობს მხოლოდ მაშინ, როდესაც საჭიროა მასთან მუშაობა.

თუ გლობალური ცვლადის სახელი ემთხვევა ლოკალური ცვლადის სახელს, მაშინ ლოკალური ცვლადის სახელი ფარავს გლობალური ცვლადის სახელს იმ ბლოკის შიგნით, რომელშიც ლოკალური ცვლადებია გამოცხადებული. ამის დემონსტრირება ხდება მოცემული პროგრამით:

```
#pragma once
int ricxvi1 = 10;
static int ricxvi2 = 20;

namespace bbb {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
```

```
using namespace System::Drawing;
. . .
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = 88;          //        ლოკალური ricxvi1 ცვლადი ფარავს გლობალურ ricxvi1 ცვლადს
int ricxvi2 = 77;          //        ლოკალური ricxvi2 ცვლადი ფარავს გლობალურ ricxvi2 ცვლადს
int ricxvi3 = ricxvi1 + ricxvi2;
label1->Text = ricxvi3.ToString();                    //        გაიცემა 165
}
```

ფორმატირებული გამოტანა

C++/CLI გარემოში გამოყენებული ფორმატის სპეციფიკატორები მოცემულია 2.12 ცხრილში.

ცხრილი 2.12. ფორმატის სპეციფიკატორები

ფორმატის სპეციფიკატორი	აღწერა
C ან c	რიცხვს აფორმატებს როგორც თანხას ადგილობრივ ვალუტაში
D ან d	რიცხვს აფორმატებს როგორც ათობით რიცხვს
E ან e	რიცხვს აფორმატებს როგორც რიცხვს ექსპონენციალური წარმოდგენით
F ან f	რიცხვს აფორმატებს როგორც წილადს
G ან g	რიცხვს აფორმატებს როგორც წილადს ან როგორც რიცხვს ექსპონენციალური წარმოდგენით
N ან n	რიცხვს აფორმატებს როგორც რიცხვს თანრიგების გამყოფებით
X ან x	მთელი რიცხვი გადაჰყავს თექვსმეტობით წარმოდგენაში

მოცემული პროგრამით ხდება ფორმატის სპეციფიკატორებთან მუშაობის დემონსტრირება.

```
{
//        Format() ფუნქციის გამოყენების დემონსტრირება
label1->Text = "";
String^ striqoni;
int mteli = 12;
double wiladi_d = 1234.56789;
float wiladi_f = 1234.56789f;
Decimal valuta = 12345;
striqoni = String::Format(L"{0:D5}\n", mteli);                    //        00012
label1->Text += striqoni;
striqoni = String::Format(L"{0:X}\n", mteli);                     //        C
label1->Text += striqoni;
striqoni = String::Format(L"{0:P3}\n", mteli);                    //        1 200,000%
label1->Text += striqoni;
striqoni = String::Format(L"{0:C3}\n", valuta);                   //        12 345,670 Lari
label1->Text += striqoni;
striqoni = String::Format(L"{0:F3}\n", wiladi_d);                 //        1234,568
label1->Text += striqoni;
```

```
striqoni = String::Format(L"{0:F3}\n", wiladi_f);          //      1234,568
label1->Text += striqoni;
striqoni = String::Format(L"{0:P3}\n", wiladi_d);          //      123 456,789%
label1->Text += striqoni;
striqoni = String::Format(L"{0:E3}\n", wiladi_d);          //      1,235E+003
label1->Text += striqoni;
striqoni = String::Format(L"{0:N3}\n", wiladi_d);          //      1 234,568
label1->Text += striqoni;
striqoni = String::Format(L"{0:G3}\n", wiladi_d);          //      1,23E+03
label1->Text += striqoni;
}
```

label1->Text = ""; ოპერატორის შესრულების შედეგად label1 კომპონენტში გამოტანილი "label1" სტრიქონი წაიშლება. საქმე ის არის, რომ როდესაც label1 კომპონენტს ფორმაზე ვათავსებთ, მის Text თვისებაში ავტომატურად გამოჩნდება "label1" სტრიქონი. მისი წაშლა შეგვიძლია Properties ფანჯარაში ან პროგრამულად ამ კომპონენტის Text თვისებისათვის ცარიელი სტრიქონის მინიჭების გზით: label1->Text = "";. თუ ამ ოპერატორს არ შევასრულებთ, მაშინ label1 კომპონენტში გამოტანილ სტრიქონს დაემატება პროგრამის მიერ გასაცემი მონაცემები.

ამრიგად, C++/CLI ვერსია იძლევა შემდეგ დამატებით შესაძლებლობებს:

- ISO/ANSI სტანდარტის მონაცემთა ყველა საბაზო ტიპი შეგვიძლია გამოვიყენოთ C++/CLI პროგრამაში.
- C++/CLI ვერსიას აქვს საკუთარი მექანიზმი კონსოლურ პროგრამაში მონაცემების კლავიატურიდან შეტანისა და ეკრანზე გამოტანისათვის.
- C++/ CLI ვერსიას აქვს save_cast ოპერატორი, რომელიც უზრუნველყოფს ტიპების უსაფრთხო გარდაქმნას.
- C++/CLI ვერსია იძლევა ჩამოთვლის შექმნის შესაძლებლობას, რომელიც არის კლასზე დაფუძნებული და უფრო მოქნილია ვიდრე ISO/ANSI C++ სტანდარტის enum განსაზღვრა (ჩამოთვლებს მე-5 თავში განვიხილავთ).

თავი 3. მმართველი ოპერატორები

არსებობს სამი კატეგორიის მმართველი ოპერატორი - ამორჩევის (if და switch), იტერაციისა (for, while, do-while და for each) და გადასვლის (break, continue, goto და return).

ამორჩევის ოპერატორი

if ოპერატორი

if ოპერატორი ამოწმებს ლოგიკურ პირობას და თუ ის ჭეშმარიტია, მაშინ შესრულდება საწყისი კოდის მითითებული ბლოკი. მისი სინტაქსია:

if (*პირობა*) { *ბლოკი1;* }
[**else** { *ბლოკი2;* }]

აქ *პირობა* არის ლოგიკური გამოსახულება, რომელიც იღებს true ან false მნიშვნელობას. თუ *პირობა* ჭეშმარიტია, მაშინ შესრულდება *ბლოკი1*, წინააღმდეგ შემთხვევაში - *ბლოკი2*. კვადრატული ფრჩხილები ნიშნავს, რომ else სიტყვის ჩაწერა აუცილებელი არ არის. ასეთ შემთხვევაში, if ოპერატორის სინტაქსი იქნება:

if (*პირობა*) { *ბლოკი1;* }

თუ პირობა იღებს true მნიშვნელობას, მაშინ შესრულდება *ბლოკი1*, წინააღმდეგ შემთხვევაში კი - პროგრამის მომდევნო ოპერატორი.

if ოპერატორის მუშაობის საჩვენებლად შევადგინოთ პროგრამა, რომელიც განსაზღვრავს კენტია რიცხვი თუ - ლუწი.

```
{
//      პროგრამა განსაზღვრავს კენტია რიცხვი თუ - ლუწი
int ricxvi;

label1->Text = " ";
ricxvi = Convert::ToInt32(textBox1->Text);
if ( ricxvi % 2 == 1 ) label1->Text = L"რიცხვი კენტია";
        else label1->Text = L"რიცხვი ლუწია";
}
```

ჩადგმული if ოპერატორი

ჩადგმულია if ოპერატორი, რომელიც მოთავსებულია სხვა if ოპერატორში. ჩადგმული if ოპერატორი იმ შემთხვევაში გამოიყენება, როდესაც მოქმედების შესასრულებლად საჭიროა რამდენიმე პირობის შემოწმება, რომელთა მითითება ერთი if ოპერატორით შეუძლებელია. ჩადგმულ if ოპერატორში else ნაწილი ყოველთვის ეკუთვნის უახლოეს if ოპერატორს. შევადგინოთ პროგრამა, რომელიც დაადგენს ორ რიცხვს შორის რომელია დადებითი:

```
{
//      პროგრამა დაადგენს ორ რიცხვს შორის რომელია დადებითი
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
```

```
if ( ricxvi1 >= 0 ) label1->Text = L"დადებითია პირველი რიცხვი";
        else    if ( ricxvi2 >= 0 ) label1->Text = L"დადებითია მეორე რიცხვი";
                        else label1->Text = L"არც ერთი რიცხვი არ არის დადებითი";
}
```

თუ პირველი if ოპერატორის პირობა ჭეშმარიტია, მაშინ label1 კომპონენტში გამოჩნდება შესაბამისი შეტყობინება და პროგრამა მუშაობას დაამთავრებს. წინააღმდეგ შემთხვევაში, შემოწმდება მეორე if ოპერატორის პირობა. თუ ის ჭეშმარიტია მაშინ label1 კომპონენტში გამოიცემა შესაბამისი შეტყობინება და პროგრამა მუშაობას დაამთავრებს. თუ არც ერთი პირობა არ არის ჭეშმარიტი, მაშინ label1 კომპონენტში გამოჩნდება შესაბამისი შეტყობინება და პროგრამა მუშაობას დაამთავრებს.

switch ოპერატორი

switch ოპერატორი საშუალებას გვაძლევს ვარიანტების სიიდან ამოვარჩიოთ საჭირო მოქმედება. მისი სინტაქსია:

switch (*გამოსახულება*)
{
case *მუდმივა1 :* ***ოპერატორების მიმდევრობა_1;***
break;
case *მუდმივა2 :* ***ოპერატორების მიმდევრობა_2;***
break;
. . .
[default : ***ოპერატორების მიმდევრობა_n;***
break;]
}

აქ ***გამოსახულება***-ს უნდა ჰქონდეს მთელრიცხვა ტიპი - char, byte, short ან int, ან სტრიქონული ტიპი - String^. გამოსახულების მნიშვნელობას არ უნდა ჰქონდეს მცურავწერტილიანი ტიპი (double, float). switch ოპერატორის case განშტოებების მუდმივები უნდა იყოს ლიტერალები და ჰქონდეთ გამოსახულებების ტიპთან თავსებადი ტიპი. ამასთან, switch ოპერატორის ერთ ბლოკში მათ არ შეიძლება ერთნაირი მნიშვნელობები ჰქონდეთ.

switch ოპერატორი შემდეგნაირად მუშაობს: გამოსახულების მნიშვნელობა მიმდევრობით შედარდება case სიაში მითითებულ თითოეულ მუდმივას. ერთ-ერთ მუდმივასთან მნიშვნელობის დამთხვევისას შესრულდება მასთან დაკავშირებული ოპერატორების მიმდევრობა.

ერთ case განშტოებასთან დაკავშირებულმა ოპერატორების მიმდევრობამ მართვა არ უნდა გადასცეს მომდევნო case განშტოების ოპერატორებს. ამის თავიდან ასაცილებლად თითოეული case განშტოების ოპერატორების მიმდევრობა break ოპერატორით უნდა დავამთავროთ.

default განშტოების შესაბამისი ოპერატორების მიმდევრობა მაშინ შესრულდება, როდესაც case განშტოებების არც ერთი მუდმივა არ შეესაბამება გამოსახულების მნიშვნელობას. default განშტოების მითითება აუცილებელი არ არის. თუ ის არ არის მითითებული და switch ოპერატორში არც ერთი მუდმივა არ შეესაბამება გამოსახულების მნიშვნელობას, მაშინ არავითარი მოქმედება არ შესრულდება და მართვა გადაეცემა switch ოპერატორის შემდეგ მოთავსებულ ოპერატორს. თუ გამოსახულების მნიშვნელობა დაემთხვა მუდმივას მნიშვნელობას, მაშინ შესრულდება ამ case განშტოებასთან ასოცირებული ოპერატორების მიმდევრობა break ოპერატორამდე. break ოპერატორი მართვას გადასცემს switch ოპერატორის შემდეგ მოთავსებულ ოპერატორს. დავწეროთ პროგრამა, რომელიც ციფრებს დააფიქსირებს 0 - 2

დიაპაზონში.

```
{
//          პროგრამა აფიქსირებს ციფრებს 0 - 2 დიაპაზონში
int cifri;

cifri = Convert::ToInt32(textBox1->Text);
switch (cifri )
{
        case 0 : label1->Text = L"თქვენ შეიტანეთ ციფრი -  " + cifri.ToString();
                    break;
        case 1 : label1->Text = L"თქვენ შეიტანეთ ციფრი -  " + cifri.ToString();
                    break;
        case 2 : label1->Text = L"თქვენ შეიტანეთ ციფრი -  " + cifri.ToString();
                    break;
        default : label1->Text = L"შეტანილი ციფრი არ არის 0 - 2 დიაპაზონში";
                     break;
}
}
```

ამ პროგრამაში switch ოპერატორის მართვა ხდება int ტიპის cifri ცვლადის მეშვეობით.

ახლა შევადგინოთ პროგრამა, სადაც switch ოპერატორის მართვა ხორციელდება Char ტიპის გამოსახულებით. ამ შემთხვევაში case განშტოების მუდმივები Char ტიპისა უნდა იყვნენ. პროგრამა გამოთვლებს ასრულებს იმის მიხედვით თუ არითმეტიკის რომელ ოპერატორს შევიტანთ.

```
{
//          კალკულატორი
int ricxvi1  = Convert::ToInt32(textBox1->Text);
int ricxvi2  = Convert::ToInt32(textBox3->Text);
Char operacia = Convert::ToChar(textBox2->Text);
switch ( operacia )
{
case '+' :          shedegi = ricxvi1 + ricxvi2;
                    label1->Text = shedegi.ToString();
                    break;
case '-' :          shedegi = ricxvi1 - ricxvi2;
                    label1->Text = shedegi.ToString();
                    break;
case '*' :          shedegi = ricxvi1 * ricxvi2;
                    label1->Text = shedegi.ToString();
                    break;
case '/' :          shedegi = ricxvi1 / ricxvi2;
                    label1->Text = shedegi.ToString();
                    break;
case '%' :          shedegi = ricxvi1 % ricxvi2;
                    label1->Text = shedegi.ToString();
                    break;
}
}
```

switch ოპერატორის ორი ან მეტი განშტოება შეიძლება დაკავშირებული იყოს ოპერატორების ერთსა და იმავე მიმდევრობასთან. ამას სხვანაირად ***ჩავარდნა*** ეწოდება. ქვემოთ მოცემული პროგრამა ახდენს ჩავარდნის დემონსტრირებას. პროგრამა განასხვავებს ქართული ანბანის ხმოვან ასოებს.

```
{
//      პროგრამით ხდება ჩავარდნის დემონსტრირება
Char simbolo;
label1->Text = " ";

simbolo = textBox1->Text[0];
switch ( simbolo )
  {
        case L'ა' :
        case L'ი' :
        case L'ო' :
        case L'ე' :
        case L'უ' : label1->Text = L"ეს ასო ხმოვანია";  break;
        default  : label1->Text = L"ეს ასო ხმოვანი არ არის";  break;
  }
}
```

თუ simbolo ცვლადი იღებს მნიშვნელობას 'ა', 'ე', 'ი', 'ო' ან 'უ', მაშინ შესრულდება label3.Text = L"ეს ასო ხმოვანია";
ოპერატორი, წინააღმდეგ შემთხვევაში კი
label3.Text = L"ეს ასო ხმოვანი არ არის";
ოპერატორი. switch ოპერატორი შეიძლება ასეც ჩავწეროთ:

```
switch ( simbolo )
{
        case L'ა' : case L'ი' : case L'ო' : case L'ე' : case L'უ' :
                label1->Text = L"ეს ასო ხმოვანია";  break;
        default : label1->Text = L"ეს ასო ხმოვანი არ არის";  break;
}
```

ჩადგმული switch ოპერატორი

ერთი switch ოპერატორი შეიძლება მოთავსებული იყოს გარე switch ოპერატორში. შიგა switch ოპერატორს ეწოდება ***ჩადგმული***. შიგა და გარე switch ოპერატორების case განშტოებების მუდმივებს შეიძლება ერთნაირი მნიშვნელობები ჰქონდეს. ეს არ გამოიწვევს კონფლიქტს პროგრამის შესრულებისას. მაგალითი:

```
{
//      პროგრამით ხდება ჩადგმული switch ოპერატორის მუშობის დემონსტრირება
Char simbolo1, simbolo2;

simbolo1 = textBox1->Text[0];
simbolo2 = textBox2->Text[0];
switch ( simbolo1 )
 {
        case L'ა' :
        label1->Text = L"ა მუდმივა ეკუთვნის გარე switch ოპერატორს";
```

```
        switch (simbolo2 )
        {
                case L'ა' :
                label2->Text = L"ა მუდმივა ეკუთვნის შიგა switch ოპერატორს";
                break;

                case L'ბ' :
                label2->Text = L"ბ მუდმივა ეკუთვნის შიგა switch ოპერატორს";
                break;
        }
        break;
case L'ბ' :
        label1->Text = L"ბ მუდმივა ეკუთვნის გარე switch ოპერატორს";
        break;
 }
}
```

იტერაციის ოპერატორი

for ოპერატორი

გამოიყენება ერთი ოპერატორის ან ოპერატორების ბლოკის მრავალჯერ შესასრულებლად. მისი სინტაქსია:

for ([*ინიციალიზატორი*]; [*პირობა*]; [*იტერატორი*]) [*ციკლის კოდი*];

ინიციალიზატორი არის გამოსახულება, რომელიც ციკლის დაწყების წინ გამოითვლება. აქ ხდება ციკლის მმართველი ცვლადის ინიციალიზება. *პირობა* არის ლოგიკური გამოსახულება, რომელიც იღებს true ან false მნიშვნელობას. ის გამოითვლება ციკლის ყოველი იტერაციის წინ. for ციკლის გამეორება ხდება მანამ, სანამ პირობა იღებს true მნიშვნელობას. თუ მან მიიღო false მნიშვნელობა, მაშინ ციკლი მთავრდება და მართვა გადაეცემა for ოპერატორის შემდეგ მოთავსებულ ოპერატორს. *იტერატორი* არის გამოსახულება, რომელიც ზრდის ან ამცირებს (ცვლის) მმართველი ცვლადის მნიშვნელობას. ის გამოითვლება ციკლის ყოველი იტერაციის შემდეგ. *ციკლის კოდი* (ციკლის ტანი) შეიძლება შეიცავდეს ერთ ან მეტ ოპერატორს.

for ციკლი შემდეგნაირად მუშაობს. ჯერ შესრულდება ციკლის მმართველი ცვლადის ინიციალიზება. შემდეგ მოწმდება პირობა. თუ ის ჭეშმარიტია, შესრულდება ციკლის კოდი. შემდეგ იცვლება ციკლის ცვლადის მნიშვნელობა. კვლავ მოწმდება პირობა და ა.შ.

შევადგინოთ პროგრამა, რომელიც ანგარიშობს 1-დან 10-მდე რიცხვების კვადრატს. პროგრამაში ციკლის მმართველი ცვლადი ერთით იზრდება.

```
{
//      პროგრამა ანგარიშობს 1-დან 10-მდე რიცხვების კვადრატს
double ricxvi, kvadrati;

label1->Text = "";
for ( ricxvi = 1; ricxvi <= 10; ricxvi++ )
  {
        kvadrati = Math::Pow(ricxvi, 2);
        label1->Text +=  ricxvi.ToString() + " - " + kvadrati.ToString() + "\n";
```

```
  }
}
```

ეს პროგრამა შეგვიძლია ასეც ჩავწეროთ:

```
{
//      პროგრამა ანგარიშობს 1-დან 10-მდე რიცხვების კვადრატს
double ricxvi, kvadrati;

label1->Text = "";
for ( ricxvi = 1; ricxvi <= 10; ricxvi++ )
  {
      kvadrati = pow(ricxvi, 2);
      label1->Text +=  ricxvi.ToString() + " - " + kvadrati.ToString() + "\n";
  }
}
```

ამ შემთხვევაში ფაილის დასაწყისში უნდა მოვათავსოთ #include <cmath> დირექტივა.

ციკლის ცვლადი შეიძლება ზრდიდეს ან ამცირებდეს თავის მნიშვნელობას ნებისმიერი სიდიდით. მოცემულ პროგრამაში ციკლის ცვლადი მცირდება 10 ერთეულით.

```
{
//      ციკლის მმართველი ცვლადი იცვლება უარყოფითი ბიჯით
double ricxvi, kvadrati;

label1->Text = " ";
for ( ricxvi = 100; ricxvi >= 1; ricxvi -= 10 )
  {
      kvadrati = Math::Pow(ricxvi, 2);
      label1->Text += ricxvi.ToString() + " - " + kvadrati.ToString() + "\n";
}
```

როგორც ვიცით, for ციკლის პირობა ციკლის დასაწყისში მოწმდება. ეს იმას ნიშნავს, რომ თუ პირობა თავიდანვე იღებს false მნიშვნელობას, მაშინ ციკლის კოდი არასოდეს არ შესრულდება, მაგალითად,

```
for ( ricxvi1 = 5; ricxvi1 < 3; ricxvi1++ )
      ricxvi2 = ricxvi1 * ricxvi1;
```

ამ ციკლის კოდი არასოდეს არ შესრულდება, რადგან პირობის გამოსახულება თავიდანვე იღებს false მნიშვნელობას.

for ციკლში ერთის ნაცვლად შეგვიძლია ორი მმართველი ცვლადის გამოყენება, მაგალითად,

```
{
int ricxvi1, ricxvi2;
label1->Text = "";

for ( ricxvi1 = 0, ricxvi2 = 10; ricxvi1 < ricxvi2; ricxvi1++, ricxvi2-- )
      label1->Text += ricxvi1.ToString() + "   " +  ricxvi2.ToString() + "\n";
}
```

აქ ინიციალიზების ორი ოპერატორი გამოყოფილია მძიმით. ასევე, ორი იტერაციული გამოსახულება გამოყოფილია მძიმით. როდესაც ციკლი იწყებს მუშაობას ორივე ცვლადს ენიჭება საწყისი მნიშვნელობები. ყოველი იტერაციის შემდეგ ricxvi1 ცვლადი მნიშვნელობა ერთით იზრდება, ricxvi2 ცვლადის მნიშვნელობა კი - ერთით მცირდება. for ციკლის მართვა

რამდენიმე მმართველი ცვლადის გამოყენებით ზოგჯერ საკმაოდ მოხერხებულია და ამარტივებს ალგორითმებს. ციკლში დასაშვებია ნებისმიერი რაოდენობის მმართველი ცვლადის გამოყენება, მაგრამ პრაქტიკულად, ორ ცვლადზე მეტი იშვიათად გამოიყენება.

for ციკლის ჩაწერისას შეგვიძლია არ მივუთითოთ ინიციალიზების, პირობის ან იტერაციის ნაწილი. მაგალითი:

```
{
//      პროგრამაში for ოპერატორი გამოყენებულია იტერაციის ნაწილის გარეშე
int ricxvi1;

label1->Text = "";
//      ამ ციკლში მითითებული არ არის იტერაციის ნაწილი
for ( ricxvi1 = 0; ricxvi1 < 10;  )
  {
        label1->Text += ricxvi1.ToString() + "\n";
        ricxvi1++;
  }
}
```

აქ არ არის მითითებული იტერაციის ნაწილი, სამაგიეროდ, მმართველი ცვლადი ერთით იზრდება ციკლის ტანში.

მოცემულ პროგრამაში გამოყენებულია for ციკლი, რომელშიც არ არის მითითებული ინიციალიზებისა და იტერაციის ნაწილები.

```
{
//      პროგრამაში for ოპერატორი გამოყენებულია ინიციალიზებისა და იტერაციის
//      ნაწილების გარეშე
int ricxvi1 = 0;

label1->Text = "";
//      ციკლი ინიციალიზატორისა და იტერატორის გარეშე
for ( ; ricxvi1 < 10;  )
  {
        label1->Text += ricxvi1.ToString() + "\n";
        ricxvi1++;
  }
}
```

აქ ricxvi1 ცვლადის ინიციალიზება ხდება ციკლამდე. ასე ძირითადად ვიქცევით მაშინ, როდესაც მმართველი ცვლადი საწყის მნიშვნელობას იღებს რაიმე რთული გამოსახულების გამოთვლის შედეგად.

თუ for ციკლის არც ერთი ნაწილი არ არის მითითებული, მაშინ მივიღებთ უსასრულო ციკლს:

```
for ( ; ; )
{
// ციკლის კოდი
}
```

უსასრულო ციკლის შესაწყვეტად შეგვიძლია break ოპერატორის გამოყენება.

for ოპერატორს შეიძლება არ ჰქონდეს, აგრეთვე, ციკლის კოდი ანუ ის იყოს ცარიელი. ეს შესაძლებელია, რადგან ნულოვანი ოპერატორი სინტაქსურად დასაშვებია. ქვემოთ მოცემულია პროგრამა, რომელშიც ასეთი ციკლი გამოიყენება 1-დან 5-მდე რიცხვების შესაკრებად.

```
{
//      პროგრამაში გამოყენებულია for ოპერატორი ციკლის კოდის გარეშე
int ricxvi, jami = 0;

label1->Text = "";
for ( ricxvi = 1; ricxvi <= 5; jami += ricxvi++ );                    //      ამ ციკლში კოდი არ არის
label1->Text = jami.ToString();
}
```

პროგრამიდან ჩანს, რომ ჯამის გამოთვლა ხდება for ოპერატორის სათაურში. შედეგად, ციკლის კოდი საჭირო არ არის. რაც შეეხება გამოსახულებას jami += ricxvi++; ის შემდეგნაირად მუშაობს. jami ცვლადს ენიჭება მისსავე მნიშვნელობას დამატებული ricxvi ცვლადის მნიშვნელობა. შემდეგ, ricxvi მნიშვნელობა ერთით იზრდება. ეს ოპერატორი შემდეგი ორი ოპერატორის ანალოგიურია:

```
jami = jami + ricxvi;
ricxvi++;
```

ხშირად მმართველი ცვლადი გამოიყენება მხოლოდ for ოპერატორის შიგნით. ასეთ შემთხვევაში, მისი გამოცხადება ხდება ციკლის საინიციალიზაციო ნაწილში. განვიხილოთ პროგრამა, რომელიც შეკრებს პირველ ხუთ რიცხვს.

```
{
//      პროგრამაში გამოყენებულია for ოპერატორი, რომელშიც გამოცხადებულია
//      მმართველი ცვლადი
int jami = 0;

label1->Text = "";
for ( int ricxvi = 1; ricxvi <= 5; ricxvi++ )
        jami += ricxvi;
label1->Text = jami.ToString();
}
```

ricxvi მმართველი ცვლადი გამოცხადებულია for ციკლის საინიციალიზაციო ნაწილში, ამიტომ ის ხილულია მხოლოდ ამ ციკლის ფარგლებში. ციკლის გარეთ ის უხილავია. თუ ვაპირებთ მმართველი ცვლადის გამოყენებას პროგრამის სხვა ნაწილში, მაშინ ის უნდა გამოვაცხადოთ for ციკლის გარეთ.

while ოპერატორი

გამოიყენება ერთი ოპერატორის ან ოპერატორების ბლოკის მრავალჯერ შესასრულებლად. მისი სინტაქსია:

while (*პირობა*)
{
ციკლის კოდი
}

აქ *პირობა* არის ლოგიკური გამოსახულება, რომელიც გასცემს true ან false მნიშვნელობას. თუ ის იღებს false მნიშვნელობას, მაშინ ხდება ციკლიდან გამოსვლა და სრულდება მომდევნო ოპერატორი. თუ ის იღებს true მნიშვნელობას, მაშინ შესრულდება ციკლის კოდი.

ამ პროგრამას ეკრანზე გამოაქვს ქართული ანბანის ასოები.

```
{
//      პროგრამას label კომპონენტში გამოაქვს ქართული ანბანის ასოები
```

```
Char simbolo = L'ა';

label1->Text = "";
while ( simbolo <= L'ჰ' )
  {
        label1->Text += simbolo.ToString() + "\n";
        simbolo++;
  }
}
```

do-while ოპერატორი

გამოიყენება ერთი ოპერატორის ან ოპერატორების ბლოკის მრავალჯერ შესასრულებლად. for და while ციკლებისაგან განსხვავებით, რომლებშიც ჯერ მოწმდება პირობა და შემდეგ სრულდება ციკლის კოდი, do-while ციკლში ჯერ სრულდება ციკლის კოდი, შემდეგ კი შემოწმდება პირობა. მისი სინტაქსია:

do
{
ციკლის კოდი
}
while (*პირობა*)

თუ ციკლში ერთი ოპერატორი სრულდება, მაშინ ფიგურული ფრჩხილების გამოყენება აუცილებელი არ არის, მაგრამ მათ ხშირად იყენებენ do-while ციკლის წაკითხვის გაუმჯობესების მიზნით, რადგან ის ადვილად შეიძლება აგვერიოს while ციკლში.

ქვემოთ მოცემულია პროგრამა, რომელსაც label კომპონენტში გამოაქვს ქართული ანბანის ასოები 'ა'-დან 'კ'-მდე.

```
{
//      პროგრამას label კომპონენტში გამოაქვს ქართული ანბანის ასოები
Char simbolo = L'ა';

label1->Text = "";
do
  {
        label1->Text += simbolo.ToString() + "\n";
        simbolo++;
  }
while ( simbolo != L'კ' );
}
```

გადასვლის ოპერატორები

break ოპერატორი

გვაძლევს ციკლის შესრულების იძულებით შეწყვეტისა და ციკლიდან გამოსვლის საშუალებას. ამ დროს, ციკლის დანარჩენი ოპერატორები არ შესრულდება. მართვა გადაეცემა ციკლის შემდეგ მოთავსებულ ოპერატორს. ქვემოთ მოცემული პროგრამა რიცხვებს ამრავლებს

მანამ, სანამ ნამრავლი არ გახდება 30-ზე მეტი.

```
{
//      პროგრამა რიცხვებს ამრავლებს მანამ, სანამ ნამრავლი არ გახდება 30-ზე მეტი
int indexi, namravli = 1;

label1->Text = "";
for ( indexi = 1; indexi < 50; indexi++ )
  {
        namravli *= indexi;
        if ( namravli > 30 ) break;
        label1->Text += "\n" + namravli.ToString();
  }
indexi--;
label1->Text =L "ციკლი შესრულდა " + indexi.ToString() + L"-ჯერ";
}
```

ციკლი უნდა შესრულდეს 50-ჯერ, მაგრამ break ოპერატორის შესრულება იწვევს მის ვადამდე შეწყვეტას. ციკლი შესრულდება მხოლოდ ოთხჯერ.

break ოპერატორის გამოყენება შეიძლება ნებისმიერ ციკლში. თუ break ოპერატორი გამოიყენება ჩადგმული ციკლის შიგნით, მაშინ ის შეწყვეტს მხოლოდ შიგა ციკლის მუშაობას. მაგალითი:

```
{
//      პროგრამა თვლის თუ რამდენჯერ შესრულდა თითოეული ციკლი
int gare = 0, shida = 1;
label2->Text = "";

for ( int i1 = 0; i1 < 5; i1++ )                    //      გარე ციკლი
{
        gare++;
        label1->Text += L"გარე ციკლში იტერაციების რაოდენობა = " + gare.ToString();

while ( shida < 30 )                                //      შიგა ციკლი
{
//      break ოპერატორი შეწყვეტს შიგა ციკლის შესრულებას
//      თუ shida ცვლადი იღებს მნიშვნელობას 5
if ( shida == 5 ) break;
        label2->Text += shida.ToString() + "   ";
        shida++;
}
label1->Text += "\n";
}
}
```

continue ოპერატორი

საშუალებას გვაძლევს იძულებით გადავიდეთ მომდევნო იტერაციაზე. continue ოპერატორის შემდეგ მოთავსებული ოპერატორები არ შესრულდება. განვიხილოთ პროგრამა, რომელსაც label კომპონენტში გამოაქვს კენტი რიცხვები 1-დან 21-მდე.

```
{
//      პროგრამას label1 კომპონენტში გამოაქვს კენტი რიცხვები 1-დან 21-მდე
int indexi;

label1->Text = "";
for ( indexi = 1; indexi <= 21; indexi++ )
  {
        if ( ( indexi % 2 ) == 0 ) continue;                //      მომდევნო იტერაციაზე გადასვლა
        label1->Text += indexi.ToString() + "\n";
  }
}
```

while და do-while ციკლებში continue ოპერატორის გამოყენება იწვევს მართვის გადაცემას უშუალოდ იტერატორისათვის, რის შემდეგაც ციკლის შესრულება გრძელდება.

goto ოპერატორი

goto არის უპირობო გადასვლის ოპერატორი. მისი სინტაქსია:

goto *ჭდე;*

სადაც, ***ჭდე*** არის იდენტიფიკატორი, რომელსაც ორწერტილი (:) მოსდევს. ის მიუთითებს იმ ადგილს, საიდანაც უნდა გაგრძელდეს პროგრამის შესრულება. თანამედროვე დაპროგრამებაში goto ოპერატორი ნაკლებად გამოიყენება, რადგან მისი გამოყენებისას პროგრამა ხდება ნაკლებად მოდულური და სტრუქტურირებული. თუმცა მისი გამოყენება ზოგჯერ საკმაოდ მოხერხებულია. ჭდე მოქმედებს მხოლოდ მოცემული ფუნქციის შეგნით. მოცემულ პროგრამაში ნაჩვენებია თუ როგორ ხდება ციკლის ორგანიზება goto ოპერატორის გამოყენებით:

```
{
//      goto ოპერატორთან მუშაობა
int ricxvi = 1;

cikli_1:
        ricxvi++;
        if ( ricxvi < 8 ) goto cikli_1;
label1->Text = ricxvi.ToString();
}
```

ტერნარული ოპერატორი

? ოპერატორს ეწოდება ტერნარული, რადგან მას სამი ოპერანდი აქვს. მისი სინტაქსია:

გამოსახულება1* ? *გამოსახულება2* : *გამოსახულება3;

აქ *გამოსახულება1* არის ლოგიკური გამოსახულება, ხოლო *გამოსახულება2* და *გამოსახულება3* გამოსახულებებია, რომლებსაც ერთნაირი ტიპი უნდა ჰქონდეს. თავდაპირველად შეფასდება *გამოსახულება1*. თუ მისი მნიშვნელობაა true, მაშინ გაიცემა გამოსახულება*2*-ის მნიშვნელობა, წინააღმდეგ შემთხვევაში კი - გამოსახულება*3*-ის მნიშვნელობა. მაგალითად,

```
max = ricxvi1 > ricxvi2 ? ricxvi1 : ricxvi2;
min = ricxvi1 < ricxvi2 ? ricxvi1 : ricxvi2;
```

პირველი გამოსახულება ricxvi1 და ricxvi2 რიცხვებიდან ამოირჩევს მაქსიმალურს, მეორე კი - მინიმალურს.

მოცემულ პროგრამაში ხდება ორი რიცხვის გაყოფა ისე, რომ თავიდან ავიცილოთ ნულზე გაყოფა.

```
{
//      50 იყოფა -3-დან 5-მდე მოთავსებულ რიცხვებზე ისე, რომ ხდება 0-ზე გაყოფის
//      თავიდან აცილება
int shedegi;

label1->Text = "";
for ( int indexi = -3; indexi < 5; indexi++ )
  {
        shedegi = indexi != 0 ? 50 / indexi : 0;
        if ( indexi != 0 ) label1->Text += indexi.ToString() + "   " + shedegi.ToString() + "\n";
  }
}
```

მაგალითები

შევადგინოთ პროგრამა, რომელიც შეკრებს რიცხვებს მითითებულ დიაპაზონში.

```
{
//      მითითებულ დიაპაზონში რიცხვების შეკრების პროგრამა
int index, min, max, jami = 0;

min = Convert::ToInt32(textBox1->Text);                 //      საწყისი რიცხვი
max = Convert::ToInt32(textBox2->Text);                 //      საბოლოო რიცხვი
for ( index = min; index <= max; index++ )
        jami += index;
label1->Text = jami.ToString();
}
```

შევადგინოთ ფაქტორიალის გამოთვლის პროგრამა.

```
{
//      ფაქტორიალის გამოთვლის პროგრამა
int index, ricxvi, shedegi = 1;

ricxvi = Convert::ToInt32(textBox1->Text);
for ( index = 1; index <= ricxvi; index++ )
        shedegi *= index;
label1->Text = shedegi.ToString();
}
```

შევადგინოთ პროგრამა, რომელიც გამოთვლის მოცემული მწკრივის მნიშვნელობას

$$y = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}.$$

```
{
//      მწკრივის გამოთვლის პროგრამა
double shedegi = 0, x;
int minusi = -1;
```

```
x  = Convert::ToDouble(textBox1->Text);
for ( int indexi = 1; indexi <= 9; indexi += 2 )
        {
                minusi *= -1;
                shedegi += minusi * Math::Pow(x, indexi) / indexi;
        }
label1->Text = Math::Round(shedegi, 2).ToString();
}
```

თავი 4. მასივი და ბიტობრივი ოპერაცია

ISO/ANSI მასივი

მასივი არის ერთი ტიპის მქონე ობიექტების კოლექცია (ერთობლიობა). ის გამოიყენება ერთნაირი ტიპის მქონე რამდენიმე ცვლადის ან ობიექტის შესანახად. მასივის ელემენტთან მიმართვისათვის უნდა მივუთითოთ მასივის სახელი და ელემენტის ინდექსი. მასივი შეიძლება იყოს როგორც ერთგანზომილებიანი, ისე მრავალგანზომილებიანი. იგი მრავალი ამოცანის გადასაწყვეტად გამოიყენება, რადგან ერთი ტიპის მქონე მონაცემების დაჯგუფების მოხერხებული საშუალებაა. მასივში შეგვიძლია შევინახოთ თანამშრომლების გვარები, ხელფასი, ასაკი და ა.შ.

მასივი მიმართვითი ტიპის ობიექტია. მონაცემები მასივში ისეა ორგანიზებული, რომ ადვილია ამ მონაცემებით მანიპულირება. სწორედ ესაა მასივის მთავარი დანიშნულება.

ერთგანზომილებიანი მასივი

ერთგანზომილებიან მასივში კონკრეტული ელემენტის მდებარეობა ერთი ინდექსით განისაზღვრება. ერთგანზომილებიანი მასივის სინტაქსია:

ტიპი მასივის_სახელი [***ზომა***];

აქ *ტიპი* განსაზღვრავს მასივის ტიპს ანუ მისი თითოეული ელემენტის ტიპს. *ზომა* არის მასივში ელემენტების რაოდენობა. მასივი რეალიზებულია ობიექტების სახით. ქვემოთ მოცემულ სტრიქონში იქმნება int ტიპის მასივი, რომელიც 10 ელემენტისაგან შედგება და მიმართვითი ტიპის masivi ცვლადს ენიჭება მასზე მიმართვა:

```
int masivi[10];
```

masivi ცვლადი ინახავს მიმართვას მეხსიერების უბანზე (ანუ ამ უბნის მისამართს), რომელიც მასივს გამოეყო. გამოყოფილი მეხსიერების ზომა საკმარისია მასში 10 ცალი int ტიპის ელემენტის მოსათავსებლად. რადგან int ტიპის ცვლადი 4 ბაიტს (32 ბიტს) იკავებს, ამიტომ masivi მასივისათვის მეხსიერებაში სულ 4x10=40 ბაიტი გამოიყოფა.

მასივის ცალკეულ ელემენტთან მიმართვა ხორციელდება ინდექსის გამოყენებით. მისი მეშვეობით ვუთითებთ ელემენტის პოზიციას მასივის ფარგლებში. ნებისმიერი მასივის პირველ ელემენტს აქვს ნულოვანი ინდექსი. რადგან masivi მასივი 10 ელემენტისაგან შედგება, ამიტომ ამ მასივის ინდექსის მნიშვნელობები მოთავსებული იქნება 0÷9 დიაპაზონში. შედეგად, masivi მასივის პირველი ელემენტი აღინიშნება ასე - masivi[0], უკანასკნელი კი - ასე masivi[9].

ქვემოთ მოცემულ პროგრამაში სრულდება masivi მასივის შევსება ციფრებით 0-დან 9-მდე და შემდეგ მათი გამოტანა label კომპონენტში.

```
{
//      მასივის შევსება რიცხვებით და label კომპონენტში მათი გამოტანა
int masivi[10];                 //      masivi მასივის გამოცხადება
int indexi;
label1->Text = "";
//      masivi მასივის შევსება რიცხვებით 0-დან 9-მდე
for ( indexi = 0; indexi < 10; indexi++ )
        masivi[indexi] = indexi;
//      masivi მასივის ელემენტების გამოტანა label1 კომპონენტში
for ( indexi = 0; indexi < 10; indexi++ )
        label1->Text += masivi[indexi].ToString() + " ";
```

```
}
```

ერთგანზომილებიანი მასივის ინიციალიზება

მასივის ინიციალიზება შესაძლებელია მისი შექმნისთანავე. ერთგანზომილებიანი მასივის ინიციალიზების სინტაქსია:

ტიპი მასივის_სახელი **[ ] { *სიდიდე1, სიდიდე2, ..., სიდიდეN*};**

აქ საწყისი მნიშვნელობები მოცემულია: *სიდიდე1*, ... *სიდიდეN* სიდიდეებით. მასივის ელემენტებს მნიშვნელობები ენიჭება რიგრიგობით, მარცხნიდან მარჯვნივ, დაწყებული ნულოვანი ინდექსის მქონე ელემენტიდან. ქვემოთ მოცემულია პროგრამა, რომელშიც სრულდება მასივის ინიციალიზება. პროგრამა პოულობს masivi მასივის მაქსიმალურ და მინიმალურ ელემენტებს.

```
{
//      პროგრამა პოულობს მასივის მინიმალურ და მაქსიმალურ ელემენტებს
//      masivi მასივის ინიციალიზება
int masivi[] = { 2, -5, 50, 15, -20, 25, 12, -1, 120, 10 };
int indexi, max, min;

max = min = masivi[0];
for ( indexi = 1; indexi < 10; indexi++ )
{
        if ( masivi[indexi] > max ) max = masivi[indexi];
        if ( masivi[indexi] < min ) min = masivi[indexi];
}
label1->Text = L" მაქსიმალური მნიშვნელობა = "  + max.ToString() +
  L"\n  მინიმალური მნიშვნელობა = " + min.ToString();
}
```

მასივის ინდექსის მნიშვნელობა არ უნდა გასცდეს ამ მასივის ინდექსის მაქსიმალურ მნიშვნელობას, წინააღმდეგ შემთხვევაში, აღიძვრება შესაბამისი შეცდომა.

ჩვენ განვიხილეთ მთელრიცხვა მასივის მაგალითი. ბუნებრივია, შეგვიძლია შევქმნათ წილადი, ლოგიკური და ა.შ. ტიპის მასივიც:

```
double wiladi_masivi[] = { 1.2, 3.9, 4.7, 6.0, 8.6 };
bool logikuri_masivi[] { true, false, true };
```

ორგანზომილებიანი მასივი

ორგანზომილებიან მასივში კონკრეტული ელემენტის მდებარეობა განისაზღვრება ორი ინდექსით. მასივის გამოცხადებისას პირველი ინდექსი მიუთითებს სტრიქონების რაოდენობას, მეორე კი - სვეტების რაოდენობას. მაგალითად, ორგანზომილებიანი მთელრიცხვა cxrili მასივი ზომით 5x4 შეგვიძლია შემდეგნაირად გამოვაცხადოთ:

```
int cxrili[5][4];
```

რიცხვი 5 განსაზღვრავს სტრიქონების რაოდენობას, რიცხვი 4 - კი სვეტების რაოდენობას. მეხსიერება გამოიყოფა 5x4=20 რიცხვისთვის. რადგან მთელი რიცხვი იკავებს 4 ბაიტს, ამიტომ cxrili მასივისთვის მეხსიერებაში გამოიყოფა 4x5x4=80 ბაიტი.

ორგანზომილებიანი მასივის კონკრეტულ ელემენტთან მიმართვისათვის აუცილებელია ორი ინდექსის მითითება:

```
cxrili[3][2] = 15;
```

აქ რიცხვი 3 არის სტრიქონის ნომერი, რიცხვი 2 - კი სვეტის ნომერი.

ქვემოთ მოცემულია პროგრამა, რომელიც ჯერ ავსებს ორგანზომილებიან მასივს მთელი რიცხვებით, შემდეგ კი ისინი label კომპონენტში გამოაქვს:

```
{
//      ორგანზომილებიანი მასივის შევსება რიცხვებით და
//      label კომპონენტში მათი გამოტანა
int cxrili[5][4];
int striqoni, sveti;

label1->Text = "";
for ( striqoni = 0; striqoni < 5; striqoni++ )
        for ( sveti = 0; sveti < 4; sveti++ )
                cxrili[striqoni][sveti] = ( striqoni * 4 ) + ( sveti + 1 );
for ( striqoni = 0; striqoni < 5; striqoni++ )
{
for ( sveti = 0; sveti < 4; sveti++ )
        label1->Text += cxrili[striqoni][sveti].ToString() + " ";
label1->Text += "\n";
}
}
```

ორგანზომილებიანი მასივის ინიციალიზება

ორგანზომილებიანი მასივების ინიციალიზებისათვის თითოეული განზომილების მნიშვნელობების სია უნდა მოვათავსოთ ცალკე ბლოკში, რომელიც, თავის მხრივ, ფიგურულ ფრჩხილებში უნდა იყოს მოთავსებული. მაგალითად,

```
int cxrili[5][2] = {
                        {1,1},
                        {2,4},
                        {3,9},
                        {4,16},
                        {5,25}
                };
```

აქ ყოველი შიგა ბლოკი მნიშვნელობას ანიჭებს შესაბამისი სტრიქონის ელემენტებს. სტრიქონის ფარგლებში პირველი მნიშვნელობა მოთავსდება სტრიქონის ნულოვან ელემენტში, მეორე მნიშვნელობა - პირველ ელემენტში და ა.შ. ბლოკები ერთმანეთისაგან მძიმეებით გამოიყოფა. ქვემოთ მოცემულ პროგრამაში ხდება ორგანზომილებიანი მასივის ინიციალიზება და მისი ელემენტების ჯამის გამოთვლა.

```
{
//      პროგრამა გამოთვლის ორგანზომილებიანი მასივის ელემენტების ჯამს
int striqoni, sveti, jami = 0;

label1->Text = "";
// masivi მასივის ინიციალიზება
int masivi[5][2] = {
                        {1,1},
                        {2,4},
                        {3,9},
                        {4,16},
```

```
                        {5,25}
                };
// მასივის ელემენტებისა და მათი ჯამის ეკრანზე გამოტანა
for ( striqoni = 0; striqoni < 5; striqoni++ )
{
        for ( sveti = 0; sveti < 2; sveti++ )
                {
jami += masivi[striqoni][sveti];
label1->Text += masivi[striqoni][sveti].ToString() + " ";
                }
        label1->Text += "\n";
}
label2->Text = jami.ToString();
}
```

მიმთითებელი

მიმთითებელი **(*ცვლადი-მიმთითებელი*)** არის ცვლადი, რომელიც შეიცავს რაიმე ტიპის სხვა ცვლადის მისამართს. მას აქვს სახელი და ტიპი. ტიპი განსაზღვრავს თუ რა ტიპის მონაცემებზე მიუთითებს მიმთითებელი. მიმთითებელს, რომელიც double ტიპის მთელ რიცხვზე მიუთითებს მეხსიერებაში „double ტიპზე მიმთითებელი“ ეწოდება. თვითონ მიმთითებელი ყოველთვის მთელი რიცხვია.

მიმთითებლები გამოიყენება მასივის ელემენტებზე ოპერაციების შესრულებისას (ხშირად ოპერაციები მასივის ელემენტებზე უფრო სწრაფად სრულდება მიმთითებლების გამოყენებით) და ფუნქციიდან ფუნქციის გარეთ განსაზღვრულ მასივებთან მიმართვისას. ბოლოს, რაც მთავარია, დინამიკურად გამოყოფილ მეხსიერებასთან მუშაობის ერთადერთი გზაა მიმთითებლის გამოყენება.

მიმთითებლის გამოცხადებისას ტიპის სახელის შემდეგ ან ცვლადის სახელის წინ * სიმბოლო უნდა დავწეროთ:

```
double* pricxvi1;
```

ან

```
double *pricxvi2;
```

აქ, pricxvi1 არის რაიმე double ტიპის რიცხვზე მიმთითებელი ანუ pricxvi1 უნდა შეიცავდეს რაიმე double ტიპის რიცხვის მისამართს.

C++ ენაში არსებობს შეთანხმება, რომლის თანახმად მიმთითებლის სახელი p ასოთი იწყება. შედეგად, ადვილდება პროგრამაში მიმთითებლის გამოყოფა.

მიმთითებლის გამოყენების ძირითადი არსი ისაა, რომ შესაძლებელია იმ მონაცემებთან მიმართვა, რომელებზეც ის მიუთითებს. ეს ხორციელდება *** ოპერაციის (*განსახელების ოპერაცია, ირიბი მიმართვის ოპერაცია*)** საშუალებით. მაგალითი:

```
{
//      * ოპერაციასთან მუშაობა
int ricxvi1 = 25, ricxvi2;
int* pricxvi1 = &ricxvi1;
*pricxvi1 *= 2;                 //      ricxvi1 = 50
(*pricxvi1)++;                  //      ricxvi1 = 51
label1->Text = (*pricxvi1).ToString();
```

```
ricxvi2 = --*pricxvi1;          //      ricxvi2 = 50
label2->Text = ricxvi2.ToString();
}
```

ამ პროგრამის მეორე სტრიქონში ხდება pricxvi1 მიმთითებლის გამოცხადება და მისთვის ricxvi1 ცვლადის მისამართის მინიჭება. & ოპერატორი გასცემს შესაბამისი ცვლადის მისამართს, ჩვენს შემთხვევაში - ricxvi1 ცვლადის მისამართს.

მომდევნოა *pricxvi1 *= 2; სტრიქონი. აქ pricxvi1 მისამართით მოთავსებული რიცხვი გამრავლდება 2-ზე და შედეგი ჩაიწერება ამავე მისამართით.

როგორც ვხედავთ, ცვლადის გამოცხადებისას * სიმბოლო მიუთითებს, რომ ხდება მიმთითებლის გამოცხადება, ხოლო შემდეგ, პროგრამაში ის მიუთითებს იმას, რომ უნდა მოხდეს ამ მისამართით მოთავსებულ მონაცემთან მიმართვა.

მომდევნოა (*pricxvi1)++; სტრიქონი. აქ pricxvi1 მისამართით მოთავსებული რიცხვი ერთით გაიზრდება. შემდეგ, ხდება ამ რიცხვის გამოტანა label1 კომპონენტში.

ricxvi2 = --*pricxvi1; სტრიქონში pricxvi1 მისამართით მოთავსებული რიცხვი ერთით შემცირდება და მიღებული შედეგი ricxvi2-ს მიენიჭება.

მიმთითებლის ინიციალიზება

არაინიციალიზებული მიმთითებლის გამოყენება არის შეცდომის წყარო. არაინიციალიზებულია მიმთითებელი, რომელიც არ შეიცავს ცვლადის მისამართს. როდესაც მიმთითებლის ინიციალიზებას ვახდენთ სხვა ცვლადის მისამართით, ეს ცვლადი გამოცხადებული უნდა იყოს მიმთითებლის გამოცხადებამდე.

მიმთითებლის გამოცხადებისას არ არის აუცილებელი მისი ინიციალიზება მისამართით. მაგალითი:

```
int* pricvxi1 = 0;
int* pricxvi2;
```

pricvxi1 მიმთითებელი არაფერზე არ მიუთითებს, რადგან 0 მისამართით არ შეიძლება ობიექტების მოთავსება. pricxvi2 მიმთითებელიც არაფერზე არ მიუთითებს.

იმისათვის, რომ შევამოწმოთ შეიცავს თუ არა მიმთითებელი კონკრეტულ მისამართს, უნდა გამოვიყენოთ შედარების ოპერატორი:

```
if ( pricxvi1 == 0 ) label1->Text = L"pricxvi მიმთითებელი მისამართს არ შეიცავს";
```

ეს ოპერატორი შეგვიძლია ასეც ჩავწეროთ:

```
if ( !pricxvi2 ) label1->Text = L"pricxvi მიმთითებელი მისამართს არ შეიცავს";
```

მიმთითებლის ინიციალიზების მაგალითი:

```
{
int ricxvi = 25;
int* pricxvi;
pricxvi = &ricxvi;
label1->Text = ((int)pricxvi).ToString();          //      2812604
label2->Text = pricxvi->ToString();                //      25
}
```

პროგრამის მესამე სტრიქონში ხდება pricxvi მიმთითებლის ინიციალიზება და მას ენიჭება ricxvi ცვლადის მისამართი.

& ოპერატორი

ცვლადის მისამართის მისაღებად გამოიყენება **& ოპერატორი (*მისამართის მიღების***

ოპერატორი). მიმთითებლისთვის ცვლადის მისამართის მინიჭების მაგალითი:

```
{
int ricxvi = 25;
int* pricxvi;
pricxvi = &ricxvi;                  //      pricxvi მიმთითებელს ენიჭება ricxviცვლადის მისამართი
label1->Text = ((int)pricxvi).ToString();           //      2812604 - ricxviცვლადის მისამართი
label2->Text = pricxvi->ToString();                 //      25
label3->Text = (&ricxvi)->ToString();               //      25
label4->Text = (*pricxvi).ToString();               //      25
}
```

აქ label1 კომპონენტში გამოჩნდება pricxvi მიმთითებელში მოთავსებული მისამართი. მისამართი გამოჩნდება თვლის ათობით სისტემაში - 2812604. label2, label3 და label4 კომპონენტებში გამოჩნდება pricxvi მისამართით მოთავსებული მნიშვნელობა.

მიმთითებელი შეიძლება მიმართავდეს ნებისმიერი ტიპის მონაცემს და ასევე შესაძლებელია ნებისმიერი ტიპის მონაცემის მისამართის მიღება. განვიხილოთ რამდენიმე მაგალითი.

მოყვანილ პროგრამაში მიმთითებელი მიმართავს წილადს:

```
{
double ricxvi = 25.75;
double* pricxvi;
pricxvi = &ricxvi;                                  //      pricxvi არის წილადზე მიმთითებელი
label1->Text = ((int)pricxvi).ToString();           //      1501824 - ricxviცვლადის მისამართი
label2->Text = pricxvi->ToString();                 //      25.75
label3->Text = (&ricxvi)->ToString();               //      25.75
label4->Text = (*pricxvi).ToString();               //      25.75
}
```

მოყვანილ პროგრამაში მიმთითებელი მიმართავს სიმბოლოს:

```
{
Char simbolo = L'ა';
Char* pchar = &simbolo;             //      pchar არის სიმბოლოზე მიმთითებელი
label1->Text = ((int)pchar).ToString();             //      3112604 – simbolo ცვლადის მისამართი
label2->Text = pchar->ToString();                   //      ა
label3->Text = (&simbolo)->ToString();              //      ა
label4->Text = (*pchar).ToString();                 //      ა
}
```

მოყვანილ პროგრამაში მიმთითებელი მიმართავს სტრიქონს:

```
{
String^ striqoni = L"რომან სამხარაძე";
String^* pstriqoni;                         //      pstriqoni არის სტრიქონზე მიმთითებელი
pstriqoni = &striqoni;

label1->Text = ((int)pstriqoni).ToString();         //      1240742 - striqoni ცვლადის მისამართი
label2->Text = (*pstriqoni)->ToString();            //      რომან სამხარაძე
}
```

მიმთითებელი და მასივი

მასივის სახელი ზოგჯერ იქცევა მიმთითებლის მსგავსად. ხშირად, როდესაც ვიყენებთ ერთგანზომილებიანი მასივის სახელს, ის ავტომატურად გარდაიქმნება ამ მასივის პირველ ელემენტზე მიმთითებლად. ეს არ ეხება იმ შემთხვევას, როდესაც მასივის სახელი არის sizeof ოპერაციის ოპერანდი.

თუ გვაქვს შემდეგი გამოცხადებები:

```
int* pmasivi;
int masivi[10];
```

მაშინ შეგვიძლია დავწეროთ ასეთი მინიჭების ოპერატორი:

```
pmasivi = masivi;
```

ეს მინიჭება masivi მასივის პირველი ელემენტის მისამართს მიანიჭებს pmasivi მიმთითებელს. მასივის სახელის გამოყენება თავისთავად ნიშნავს ამ მასივის მისამართზე მიმართვას. თუ მასივის სახელს ვიყენებთ ინდექსთან ერთად, მაშინ ეს ნიშნავს მიმართვას იმ ელემენტის მნიშვნელობასთან, რომელიც შეესაბამება ინდექსის მნიშვნელობას. ამიტომ თუ გვინდა, რომ მიმთითებელში შევინახოთ მასივის ელემენტის მისამართი, უნდა გამოვიყენოთ მისამართის მიღების ოპერაცია:

```
pmasivi = &masivi[2];
```

აქ pmasivi მიმთითებელს მიენიჭება masivi მასივის რიგით მესამე ელემენტის მისამართი (რომლის ინდექსია 2). მოცემული პროგრამით ხდება ზემოთ ნათქვამის დემონსტრირება.

```
{
//      მასივზე მიმთითებელთან მუშაობა
int* pmasivi;
int masivi[10] = { 1, 2, 3, 4, 5 };

pmasivi = masivi;
label1->Text = pmasivi->ToString();          //      გამოჩნდება 1
pmasivi = &masivi[2];
label2->Text = pmasivi->ToString();          //      გამოჩნდება 3
}
```

როგორც ვხედავთ, label1 კომპონენტში გამოჩნდება მასივის პირველი ელემენტის მნიშვნელობა (მისი ინდექსია 0). შემდეგ, label2 კომპონენტში გამოჩნდება მასივის ელემენტი, რომლის ინდექსია 2.

მიმთითებლის არითმეტიკა

მიმთითებლებზე შეგვიძლია შევასრულოთ შეკრების, გამოკლებისა და შედარების ოპერაციები. მიმთითებლების არითმეტიკა არაცხადად გულისხმობს, რომ მიმთითებელი მიუთითებს მასივზე და არითმეტიკის ოპერაციები სრულდება მისამართებზე, რომელიც მიმთითებელშია მოთავსებული. დავუშვათ, შევასრულეთ მინიჭების ოპერატორი:

```
pmasivi = &masivi[3];
```

ამ შემთხვევაში, pmasivi+1 მიმართავს masivi[4] ელემენტს. ამიტომ, (pmasivi + 1)-ის ნაცვლად შეგვიძლია ჩავწეროთ pmasivi++;

ეს ოპერატორი pmasivi მიმთითებელში მოთავსებულ მისამართს ზრდის ბაიტების იმ რაოდენობით, რომელსაც იკავებს masivi მასივის თითოეული ელემენტი. ზოგად შემთხვევაში, pmasivi + n გამოსახულება, სადაც n ნებისმიერი მთელი რიცხვია, უმატებს n*sizeof(int) მნიშვნელობას pmasivi მიმთითებელში მოთავსებულ მისამართს. როგორც ვხედავთ, სრულდება მასშტაბირება ტიპის მიხედვით. მაგალითი:

```
{
```

```
int* pmasivi;
int masivi[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 1  };

pmasivi = masivi;
pmasivi = &masivi[2];
*pmasivi++;      //  მიმართვა მოხდება მასივის მომდევნო, მესამე ინდექსის მქონე  ელემენტთან - 4
label1->Text = (*pmasivi).ToString();
}
```

თუ pmasivi = &masivi[0]; , მაშინ ეკვივალენტურია *pmasivi და masivi[0]; (*pmasivi+1) და masivi[1]; (*pmasivi+2) და masivi[2] და ა.შ. ჩანაწერები.

იგივე ეხება double, Char და ა.შ. ტიპის მასივებს. double ტიპის შემთხვევაში გვექნება n*sizeof(double) და მისამართის ინკრემენტის ან დეკრემენტის შემთხვევაში ის 8 ერთეულით გაიზრდება ან შემცირდება. Char ტიპის შემთხვევაში გვექნება n*sizeof(Char) და მისამართის ინკრემენტის ან დეკრემენტის შემთხვევაში ის 2 ერთეულით გაიზრდება ან შემცირდება.

მაგალითი.

```
{
int ricxvi = 25;
int* pricxvi;
pricxvi = &ricxvi;
label1->Text = ((int)pricxvi).ToString();          //      1240742
label2->Text = (*pricxvi).ToString();              //      25
pricxvi++;
label3->Text = ((int)pricxvi).ToString();          //      1240746
pricxvi += 2;
label4->Text = ((int)pricxvi).ToString();          //      1240754
}
```

აქ label1 კომპონენტში გამოგვაქვს ricxvi ცვლადის მისამართი, რომელიც pricxvi მიმთითებელშია მოთავსებული. label2 კომპონენტში გამოგვაქვს pricxvi მისამართით მოთავსებული მნიშვნელობა. შემდეგ, სრულდება ინკრემენტის ოპერაცია pricxvi მიმთითებლის მიმართ. შედეგად, მისამართი 4 ერთეულით იზრდება, რადგან int ტიპი 4 ბაიტს იკავებს მეხსიერებაში. მისამართის ახალი მნიშვნელობა გამოგვაქვს label3 კომპონენტში. შემდეგ, მისამართს ემატება რიცხვი 2, ამიტომ ის გაიზრდება 2 x 4 = 8 ბაიტით.

მაგალითი.

```
{
double wiladi = 125.75;
double* pwiladi = &wiladi;
label1->Text = ((int)pwiladi).ToString();          //      1240740
label2->Text = (*pwiladi).ToString();              //      125.75
pwiladi++;
label3->Text = ((int)pwiladi).ToString();          //      1240748
pwiladi += 2;
label4->Text = ((int)pwiladi).ToString();          //      1240764
}
```

აქ label1 კომპონენტში გამოგვაქვს wiladi ცვლადის მისამართი, რომელიც pwiladi მიმთითებელშია მოთავსებული. label2 კომპონენტში გამოგვაქვს pwiladi მისამართით მოთავსებული მნიშვნელობა. შემდეგ, სრულდება ინკრემენტის ოპერაცია pwiladi მიმთითებლის მიმართ. შედეგად, მისამართი 8 ერთეულით იზრდება, რადგან double ტიპი 8 ბაიტს იკავებს

მეხსიერებაში. მისამართის ახალი მნიშვნელობა გამოგვაქვს label3 კომპონენტში. შემდეგ, მისამართს ემატება რიცხვი 2, ამიტომ ის გაიზრდება 2 x 8 = 16 ბაიტით.

მაგალითი.

```
{
Char simbolo = L'რ';
Char* pchar = &simbolo;
label1->Text = ((int)pchar).ToString();          //      1240740
label2->Text = (*pchar).ToString();              //      125.75
pchar++;
label3->Text = ((int)pchar).ToString();          //      1240742
pchar += 2;
label4->Text = ((int)pchar).ToString();          //      1240746
}
```

აქ label1 კომპონენტში გამოგვაქვს simbolo ცვლადის მისამართი, რომელიც pchar მიმთითებელშია მოთავსებული. label2 კომპონენტში გამოგვაქვს pchar მისამართით მოთავსებული მნიშვნელობა. შემდეგ, სრულდება ინკრემენტის ოპერაცია pchar მიმთითებლის მიმართ. შედეგად, მისამართი 2 ერთეულით იზრდება, რადგან Char ტიპი 2 ბაიტს იკავებს მეხსიერებაში. მისამართის ახალი მნიშვნელობა გამოგვაქვს label3 კომპონენტში. შემდეგ, მისამართს ემატება რიცხვი 2, ამიტომ ის გაიზრდება 2 x 2 = 4 ბაიტით.

მიმთითებელი და მრავალგანზომილებიანი მასივი

გამოვაცხადოთ ორგანზომილებიანი masivi მასივი და pmasivi მიმთითებელი. მიმთითებელს მივანიჭოთ ამ მასივის პირველი ელემენტის მისამართი:

```
int masivi[5][3];
int* pmasivi;
pmasivi = &masivi[0][0];
```

შედეგად, მიმთითებელი მიმართავს მასივის პირველ ელემენტს. შეგვიძლია, მიმთითებელს მივანიჭოთ masivi მასივის პირველი სტრიქონის მისამართი:

```
pmasivi = masivi[0];
```

რადგან, მიმთითებლის ტიპია int*, ხოლო მასივის ტიპი - int[5][3], ამიტომ დაუშვებელია ასეთი მინიჭება:

```
pmasivi = masivi;       //      შეცდომა!
```

იმისათვის, რომ მიმთითებელში მოვათავსოთ ორგანზომილებიანი მასივის მისამართი, მისი ტიპი უნდა იყოს int*[3]:

```
int masivi[5][3];
int (*pmasivi)[3];
```

ამ შემთხვევაში, დასაშვებია მინიჭება pmasivi = masivi; :

```
int masivi[5][3];
int (*pmasivi)[3];
pmasivi = masivi;
```

ორგანზომილებიან masivi მასივის ელემენტთან მიმართვა ორი გზით შეიძლება:

1. მასივის სახელისა და ორი ინდექსის მითითებით - masivi[striqoni][sveti];
2. მასივის სახელის გამოყენება როგორც მიმთითებელი - *(*(masivi + striqoni)+sveti).

ორივე გზა ეკვივალენტურია. რაც შეეხება, მეორე ჩანაწერს, აქ *(masivi + striqoni) არის striqoni სტრიქონის პირველი ელემენტის მისამართი, ხოლო *(masivi + striqoni)+sveti კი - striqoni სტრიქონის ელემენტი, რომლის წანაცვლებაა sveti. ამიტომ, *(*(masivi + striqoni)+sveti) გამოსახულება მიმართავს masivi მასივის კონკრეტულ ელემენტს.

მიმართვა

მიმართვა (reference, ссылка) არის სხვა ცვლადის ***ფსევდონიმი***. მას აქვს სახელი, რომელიც შეგვიძლია ცვლადის სახელის ნაცვლად გამოვიყენოთ. მიმართვის გამოცხადებისას უნდა მივუთითოთ იმ ცვლადის სახელი, რომლისთვისაც ის იქმნება. არ შეიძლება მიმართვის შეცვლა იმ მიზნით, რომ მან სხვა ცვლადს მიმართოს. მიმართვის გამოცხადებისათვის & სიმბოლო გამოიყენება. მაგალითი:

```
{
//      მიმართვასთან მუშაობა
int ricxvi = 5;
int& mimartva_ricxvi = ricxvi;                  //      მიმართვის გამოცხადება
label1->Text = mimartva_ricxvi.ToString();      //      5
label2->Text = ricxvi.ToString();               //      5
mimartva_ricxvi *= 2;
label3->Text = mimartva_ricxvi.ToString();      //      10
label4->Text = ricxvi.ToString();               //      10
}
```

აქ mimartva_ricxvi არის მიმართვა, რომელიც ricxvi მთელი ტიპის ცვლადის ფსევდონიმია. ამ გამოცხადების შემდეგ მიმართვა შეგვიძლია გამოვიყენოთ ცვლადის მაგივრად.

მიმართვა შეგვიძლია გამოვიყენოთ წილად რიცხვთან სამუშაოდ.

```
{
double ricxvi = 5.5;
double& mimartva_ricxvi = ricxvi;               //      მიმართვის გამოცხადება
label1->Text = mimartva_ricxvi.ToString();      //      5.5
label2->Text = ricxvi.ToString();               //      5.5
mimartva_ricxvi *= 3;
label3->Text = mimartva_ricxvi.ToString();      //      16.5
label4->Text = ricxvi.ToString();               //      16.5
}
```

მიმართვა შეგვიძლია გამოვიყენოთ სიმბოლოსთან სამუშაოდ.

```
{
Char simbolo = L'რ';
Char& mimartva_simbolo = simbolo;               //      მიმართვის გამოცხადება
label1->Text = mimartva_simbolo.ToString();     //      რ
label2->Text = simbolo.ToString();              //      რ
mimartva_simbolo = L'ს';
label3->Text = mimartva_simbolo.ToString();     //      ს
label4->Text = simbolo.ToString();              //      ს
}
```

მიმართვა შეგვიძლია გამოვიყენოთ სტრიქონთან სამუშაოდ.

```
{
String^ striqoni = L"რომანი";
String^& mimartva_striqoni = striqoni;                  //      მიმართვის გამოცხადება
label1->Text = mimartva_striqoni->ToString();           //      რომანი
```

```
label2->Text = striqoni->ToString();                    //    რომანი
mimartva_striqoni = L"სამხარაძე";
label3->Text = mimartva_striqoni->ToString();           //    სამხარაძე
label4->Text = striqoni->ToString();                    //    სამხარაძე
}
```

აქ იმართვა გამოიყენება სტრიქონთან სამუშაოდ.

sizeof ოპერაცია

sizeof ოპერაციის შესრულებით ვღებულობთ size_t ტიპის მთელ მნიშვნელობას. ის მიუთითებს ბაიტების რაოდენობას, რომელსაც მისი ოპერანდი იკავებს. size_t ტიპი განსაზღვრულია სტანდარტულ ბიბლიოთეკაში და ეფუძნება unsigned int საბაზო ტიპს.

მოცემულ მაგალითში გაიცემა ricxvi ცვლადის ზომა:

```
int ricxvi;
label1->Text = (sizeof ricxvi).ToString();
```

შედეგი იქნება 4, რადგან ricxvi არის int ტიპის ცვლადი.

sizeof ოპერაცია შეგვიძლია გამოვიყენოთ მასივის ელემენტის ან მთლიანად მასივის მიმართ. როდესაც მას ვიყენებთ მასივის სახელის მიმართ, მაშინ გაიცემა ბაიტების რაოდენობა, რომელსაც მთელი მასივი იკავებს. როდესაც მას ვიყენებთ მასივის ელემენტის მიმართ, მაშინ გაიცემა მოცემული ელემენტის მიერ დაკავებული ბაიტების რაოდენობა. მაგალითი:

```
{
//    sizeof ოპერაციასთან მუშაობა
double masivi [ ] = { 1, 2, 3, 4, 5 };
label1->Text = L"მთელი მასივის ზომა = " + (sizeof masivi).ToString();                    //    40
label2->Text = L"მასივის მესამე ელემენტის ზომა = " + (sizeof masivi[2]).ToString();      //     8
label3->Text = L"მასივში ელემენტების რაოდენობა = "
                                    + ( (sizeof masivi) / (sizeof masivi[0]) ).ToString();    //    5
}
```

sizeof ოპერაცია შეგვიძლია გამოვიყენოთ ტიპის მიმართაც:

```
size_t zoma = sizeof(double);
label1->Text = zoma.ToString();                    //    zoma = 8
```

მეხსიერების დინამიკური გამოყოფა

პროგრამის შესრულებისას ხშირად წამოიჭრება სხვადასხვა ტიპის ცვლადისათვის მეხსიერების უბნის დინამიკურად გამოყოფის აუცილებლობა. რადგან, დინამიკურად განაწილებული ცვლადები არ შეიძლება განისაზღვროს პროგრამის კომპილაციის დროს, ამიტომ მათ არ აქვთ სახელები პროგრამის საწყის კოდში. მათი შექმნისას მათი იდენტიფიკაცია ხდება მეხსიერების მისამართებით, რომლებიც მიმთითებლებში ინახება.

მეხსიერების გროვა

ხშირად, პროგრამის შესრულებისას, კომპიუტერის ოპერატიული მეხსიერების ნაწილი თავისუფალია. ამ გამოუყენებელ მეხსიერებას C++ სისტემაში ***გროვა*** ან ***თავისუფალი სათავსო*** ეწოდება. ამ გროვაში შეგვიძლია გამოვყოთ მეხსიერება მოცემული ტიპის ახალი ცვლადისთვის.

ამ მიზნით გამოიყენება new ოპერატორი. ის, აგრეთვე, გასცემს გამოყოფილი უბნის მისამართს. new ოპერატორის მიერ გამოყოფილ მეხსიერების უბანს ათავისუფლებს delete ოპერატორი.

გროვაში შეგვიძლია რომელიმე ცვლადისთვის გამოვყოთ მეხსიერების უბანი. შემდგომ, როდესაც ეს ცვლადი საჭირო აღარ იქნება, ეს უბანი შეგვიძლია გავათავისუფლოთ და გროვას დავუბრუნოთ. შედეგად, შესაძლებელი გახდება ამ უბნის გამოყენება ამავე პროგრამის მიერ, ოღონდ უკვე სხვა ცვლადისთვის.

გროვაში მეხსიერების გამოყოფის აუცილებლობა დგება ყოველთვის, როდესაც საჭიროა პროგრამის შესრულებისას განსაზღვრული ელემენტებისთვის მეხსიერების გამოყოფა. მაგალითად, ასეთი აუცილებლობა დგება მომხმარებლის მიერ სტრიქონების შეტანისას, რადგან ჩვენ წინასწარ არ ვიცით თითოეული სტრიქონის სიგრძე. ამიტომ, მათთვის მეხსიერების გამოყოფა უმჯობესია new ოპერატორის გამოყენებით.

მეხსიერების დინამიკური განაწილების შესაძლებლობა იძლევა მეხსიერების ეფექტურად გამოყენების შესაძლებლობას. ამ დროს, შეგვიძლია მეხსიერება გამოვყოთ მხოლოდ მაშინ, როდესაც გვჭირდება და გავათავისუფლოთ მაშინ, როდესაც აღარ გვჭირდება. მეხსიერების გათავისუფლებული უბანი შეგვიძლია სხვა ობიექტების მოსათავსებლად გამოვიყენოთ.

new და delete ოპერატორები

დავუშვათ, int ტიპის ცვლადისთვის გვინდა მეხსიერების გამოყოფა. ჩვენ შეგვიძლია გამოვაცხადოთ int ტიპზე მიმთითებელი და შემდეგ მოვითხოვოთ მისთვის მეხსიერების გამოყოფა პროგრამის შესრულების დროს. ეს ხორციელდება new ოპერატორის გამოყენებით:

```
int* pcvladi = NULL;          //    მიმთითებელი ინიციალიზებული როგორც NULL
pcvladi = new int;            //    მეხსიერების მოთხოვნა int ტიპის ცვლადისთვის
*pcvladi = 55;
```

აქ new ოპერატორი გასცემს გამოყოფილი მეხსიერების მისამართს, რომელიც pcvladi ცვლადს მიენიჭება.

შეგვიძლია ცვლადის ინიციალიზება მისთვის მეხსიერების გამოყოფისთანავე:

```
pcvladi = new int(77);
```

როდესაც დინამიკურად განაწილებული ცვლადი საჭირო აღარ არის, მაშინ შეგვიძლია მის მიერ დაკავებული მეხსიერების გათავისუფლება:

```
delete pcvladi;
```

ეს საშუალებას გვაძლევს გამოთავისუფლებული მეხსიერება გამოვიყენოთ მასში სხვა ცვლადების მოსათავსებლად. თუ delete ოპერაცია არ შევასრულეთ და შემდგომ pcvladi ცვლადს სხვა მისამართი მივანიჭეთ, შეუძლებელი გახდება მეხსიერების გათავისუფლება ან მასში მოთავსებული ცვლადის გამოყენება, რადგან მას ვეღარ მივმართავთ. ამას ***მეხსიერების გაჟონვა*** (memory leak) ეწოდება.

მეხსიერების დინამიკური განაწილება მასივებისთვის

char ტიპის მასივისთვის დინამიკურად გამოვყოთ მეხსიერება:

```
char* pstriqoni;
pstriqoni = new char[100];
```

აქ მეხსიერება 100-სიმბოლოიანი მასივისთვის გამოიყოფა. მისი მისამართი pstriqoni მიმთითებელში ინახება.

ასეთნაირად გამოყოფილი მეხსიერების გასათავისუფლებლად და გროვაში დასაბრუნებლად delete ოპერატორი უნდა გამოვიყენოთ:

```
delete [ ] pstriqoni;
```

კვადრატული ფრჩხილები მიუთითებს, რომ მასივი წაიშლება. delete ოპერაციის შემდეგ pstriqoni მიმთითებელი შეიძლება შეიცავდეს მეხსიერების რომელიმე შემთხვევითი უბნის მისამართს, რომლის გამოყენება არ შეიძლება. ამიტომ, მიმთითებელს უნდა მივანიჭოთ ნულოვანი მნიშვნელობა:

```
pstriqoni = 0;
```

დავუშვათ, გვაქვს ორგანზომილებიანი მასივი:

```
int masivi[5][3];
int (*pmasivi)[3];
```

მაშინ მისთვის შეგვიძლია გამოვყოთ მეხსიერება:

```
pmasivi = new int[5][3];
```

ანალოგიურად ვიქცევით სამგანზომილებიანი მასივისთვის:

```
int masivi[5][3][4];
int (*pmasivi)[3][4];
pmasivi = new int[5][3][4];
```

ნებისმიერი განზომილების მასივის წასაშლელად გამოიყენება delete ოპერაცია.

CLR მასივი

C++ ენის მშობლიური „გროვის“ მსგავსად CLR გარემოსაც აქვს საკუთარი მეხსიერების „გროვა“. CLR გროვა დამოუკიდებელია C++ ენის მშობლიური გროვისგან. CLR გროვაში მეხსიერების დინამიკური განაწილება განსხვავებულად სრულდება, ვიდრე C++ ენის მშობლიურ გროვაში. CLR გარემო ავტომატურად შლის (ათავისუფლებს) CLR გროვაში გამოყოფილ მეხსიერებას, რომელიც საჭირო აღარ არის. ამიტომ, CLR პროგრამაში delete ოპერატორი არ გამოიყენება მეხსიერების გათავისუფლების მიზნით. საჭიროების შემთხვევაში, CLR გარემო პერიოდულად ასრულებს გროვისთვის გამოყოფილი მეხსიერების გადაწყობას მეხსიერების დაფრაგმენტების ან გაჟონვის თავიდან აცილების მიზნით.

გროვის გასუფთავების პროცესის მართვას, რომელსაც CLR გარემო უზრუნველყოფს, ***ნაგვის შეგროვება*** (garbage collection) ეწოდება. ნაგავი არის გაუქმებული ცვლადები და ობიექტები. მათ გროვას (ერთობლიობას), რომელსაც CLR გარემო მართავს, ***მართვადი გროვა*** (garbage-collected heap) ეწოდება. სწორედ, მართვად გროვას ასუფთავებს ***ნაგვის შემგროვებელი***.

C++/CLI გარემოში მეხსიერების განაწილებისათვის (გამოყოფისათვის) გამოიყენება gcnew ოპერატორი C++ ენის მშობლიური new ოპერატორის ნაცვლად. „gc“ პრეფიქსი მიუთითებს, რომ მეხსიერების განაწილება სრულდება მართვად გროვაში.

CLR ნაგვის შემგროვებელს შუძლია წაშალოს ობიექტები და გაათავისუფლოს მეხსიერება, რომელიც მათ ეკავა მაშინ, როდესაც ისინი საჭირო აღარ არიან. CLR გარემო ***თვალყურს ადევნებს*** თითოეულ ცვლადს, რომელიც მიმართავს გროვის თითოეულ ობიექტს. როდესაც აღარ არიან ცვლადები, რომლებიც შეიცავენ მოცემული ობიექტის მისამართს, მაშინ ობიექტთან ვეღარ მოხდება მიმართვა პროგრამიდან და ამიტომ, ის შეიძლება წაიშალოს.

რადგან, ნაგვის შეგროვების პროცესი შეიძლება მოიცავდეს გროვისთვის გამოყოფილი მეხსიერების შეკუმშვას ფრაგმენტირებული და გამოუყენებელი მეხსიერების ბლოკების წაშლის მიზნით, ამიტომ გროვაში შენახული მონაცემების მისამართები შეიძლება შეიცვალოს. შედეგად, ჩვენ ვერ გამოვიყენებთ C++ ენის ჩვეულებრივ მშობლიურ მიმთითებლებს გასასუფთავებელი გროვის მიმართ, რადგან თუ მონაცემის ადგილმდებარეობა შეიცვალა, მაშინ მიმთითებელი აღარ იქნება ნამდვილი. საჭიროა მისამართის გაახლება მაშინ, როდესაც ნაგვის შემგროვებელი გროვაში ცვლის ობიექტების ადგილმდებარეობას. ეს ორი გზით მიიღწევა:

- **თვალყურის სადევნებელი დესკრიპტორის** (tracking handle, отслеживаемый дескриптор) გამოყენებით, რომელიც C++ ენის მშობლიური მიმთითებლის გარკვეული ანალოგია CLR პროგრამაში.
- **თვალყურის სადევნებელი მიმართვის** გამოყენებით, რომელიც არის C++ ენის მშობლიური მიმართვის ეკვივალენტი CLR პროგრამაში.

თვალყურის სადევნებელი დესკრიპტორი

თვალყურის სადევნებელი დესკრიპტორი (ან უბრალოდ ***დესკრიპტორი***) არის ცვლადი, რომელიც შეიცავს გროვაში მოთავსებული ობიექტის მისამართს. თვალყურის სადევნებელი დესკრიპტორი არის C++ ენის მშობლიური მიმთითებლის მსგავსი, თუმცა მისგან მნიშნელოვნად განსხვავდება. დესკრიპტორი ინახავს ობიექტის მისამართს, რომელიც ავტომატურად გაახლდება ნაგვის შემგროვებლის მიერ თუ შესაბამისი ობიექტი გადაადგილდა გროვის შეკუმშვისას. ჩვენ არ შეგვიძლია შევასრულოთ სამისამართო არითმეტიკა ასეთი მიმთითებლების მიმართ, როგორც ეს შეგვეძლო C++ ენის „მშობლიური“ მიმთითებლების მიმართ.

CLR გროვაში შექმნილ ყველა ობიექტს უნდა ჰქონდეს დესკრიპტორი. კლასების მიმართვითი ტიპის ყველა ობიექტი ინახება გროვაში. ამიტომ, ცვლადები, რომლებიც შევქმენით ახალ ობიექტებთან მიმართვისთვის, უნდა იყოს თვალყურის სადევნებელი დესკრიპტორები. მაგალითად, String კლასი არის კლასის მიმართვითი ტიპი. ამიტომ, ცვლადები, რომლებიც მიმართავენ String ობიექტებს, უნდა იყოს დესკრიპტორები. მეხსიერება კლასის ტიპის ცვლადებისათვის ავტომატურად (default) გამოიყოფა სტეკში, მაგრამ ჩვენ მათი მოთავსება შეგვიძლია, აგრეთვე გროვაში gcnew ოპერატორის გამოყენებით. აქვე შევნიშნოთ, რომ CLR გროვაში გამოცხადებული ცვლადები, CLR მიმართვითი ტიპების ჩათვლით, არ შეიძლება გამოცხადებული იყოს გლობალურ კონტექსტში.

ტიპისთვის დესკრიპტორის გამოცხადება ხდება ტიპის სახელის შემდეგ ^ სიმბოლოს მითითების გზით (^ სიმბოლოს ხშირად „ქუდს“ უწოდებენ):

```
String^ striqoni;
```

აქ განისაზღვრება striqoni ცვლადი როგორც String^ ტიპის თვალყურის სადევნებელი დესკრიპტორი. ის ინახავს String ტიპის ობიექტის მისამართს. როდესაც ჩვენ ვაცხადებთ დესკრიპტორს, ის ავტომატურად ინიციალიზდება როგორც null. ამიტომ, ის არაფერს არ მიმართავს. იმისათვის, რომ აშკარად მივანიჭოთ დესკრიპტორს null, უნდა გამოვიყენოთ nullptr საკვანძო სიტყვა:

```
striqoni = nullptr;
```

არ შეიძლება 0-ის გამოყენება null სიტყვის ნაცვლად, როგორც ეს მიღებულია C++ ენის მშობლიურ მიმთითებლებში. თუ ვახდენთ დესკრიპტორის ინიციალიზებას 0-ით, მაშინ 0 გარდაიქმნება იმ ტიპის ობიექტად, რომელსაც დესკრიპტორი მიმართავს და მასში შეინახება ამ ახალი ობიექტის მისამართი.

დესკრიპტორის ინიციალიზება შეიძლება, აგრეთვე აშკარად მისი გამოცხადებისას:

```
String^ striqoni = L"C++ დაპროგრამების ენაა";
```

აქ String ტიპის ობიექტი იქმნება CLR გროვაში, რომელიც სტრიქონს შეიცავს. ახალი ობიექტის მისამართი ინახება striqoni ცვლადში. ამ შემთხვევაში, თითოეული სტრიქონული ლიტერალის ტიპი არის const wchar_t* და არა String. L სიმბოლო მიუთითებს, რომ სტრიქონი Unicode სიმბოლოებს შეიცავს.

ქვემოთ მოცემულია მთელი ტიპის სიდიდისათვის დესკრიპტორის შექმნის მაგალითი:

```
int^ cvladi = 50;
```

აქ იქმნება cvladi დესკრიპტორი, რომლის ტიპია int და მნიშვნელობა გროვაში,

რომელზეც ეს დესკრიპტორი მიუთითებს, არის 50. იმისათვის, რომ cvladi დესკრიპტორი გამოვიყენოთ არითმეტიკის ოპერაციებში, მის წინ უნდა მოვათავსოთ * ოპერატორი:

```
{
int^ cvladi = 50;
int shedegi = 5 * ( *cvladi ) - 10;                    //        shedegi = 240
label1->Text = shedegi.ToString();
}
```

აქ *cvladi გამოსახულება მიმართავს მთელ რიცხვს, რომელიც შენახულია დესკრიპტორში მოთავსებული მისამართით. გამოთვლის შედეგი იქნება - 240. მინიჭების ეს ოპერატორი შეგვიძლია ასეც ჩავწეროთ:

```
{
int^ cvladi = 50;
int^ shedegi = 0;
shedegi = 5 * ( *cvladi ) - 10;
label1->Text = shedegi->ToString();
}
ან
{
int^ cvladi = 50;
int^ shedegi = 0;
*shedegi = 5 * ( *cvladi ) - 10;
label1->Text = shedegi->ToString();
}
```

ორივე შემთხვევაში shedegi დესკრიპტორი უნდა მიმართავდეს კონკრეტულ მნიშვნელობას, წინააღმდეგ შემთხვევაში აღიძვრება შეცდომა.

ერთგანზომილებიანი მასივი

CLR მასივი განსხვავდება C++-ის მშობლიური მასივისგან. CLR მასივისთვის მეხსიერება გამოიყოფა **მართვად გროვაში** (garbage-collected heap). მასივის ტიპი განისაზღვრება array საკვანძო სიტყვით. მასივის ელემენტების ტიპი უნდა მივუთითოთ კუთხურ ფრჩხილებში array სიტყვის შემდეგ. ამრიგად, ერთგანზომილებიანი მასივის გამოცხადების სინტაქსია:

array<*ელემენტის_ტიპი*>^ *მასივის_სახელი*;

რადგან, CLR მასივი იქმნება გროვაში, ამიტომ მასივის ცვლადები ყოველთვის თვალყურის სადევნებელი დესკრიპტორებია. მასივის გამოცხადების მაგალითი:

```
array<int>^ masivi;
```

masivi ცვლადი შეიძლება ინახავდეს მიმართვას int ტიპის ელემენტების ერთგანზომილებიან მასივზე. მასივის შექმნისას შეიძლება მისი ინიციალიზება:

```
array<double>^ masivi = { 1.2, 2.3, 3.4, 4.5, 5.6 };
```

მასივის ზომა მონაცემების რაოდენობით განისაზღვრება და ჩვენ შემთხვევაში ის 5-ის ტოლია.

ამავე გზით შეგვიძლია შევქმნათ სტრიქონების მასივიც:

```
array<String^>^ striqoni = { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
```

სტრიქონების მასივის გამოცხადება შეიძლება მისი ინიციალიზების გარეშეც:

```
array<String^>^ striqoni;
striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
```

CLR მასივი შეგვიძლია, აგრეთვე შევქმნათ gcnew ოპერატორის გამოყენებით:

```
array<int>^ masivi = gcnew array<int>(10);
```

აქ 10 არის მასივში ელემენტების რაოდენობა.

C++ ენის მშობლიური მასივის მსგავსად CLR მასივის ელემენტების ინდექსირება ნულიდან იწყება. მაგალითი:

```
{
// CLR მასივთან მუშაობა
array<int>^ masivi = gcnew array<int>(10);
for ( int indexi = 0; indexi < 10; indexi++ )
{
masivi[indexi] = 2 * ( indexi + 1 );
label1->Text += masivi[indexi].ToString() + "  ";
}
}
```

მოცემულ მაგალითში masivi მასივი ინახავს Int32 ტიპის ობიექტებს. for ციკლში ელემენტების რაოდენობა მითითებულია ლიტერალის სახით - 10. მის ნაცვლად უმჯობესია Length თვისების გამოყენება, რომელშიც ავტომატურად მოთავსთდება მასივის ელემენტების რაოდენობა. მაგალითი:

```
{
// Length თვისებასთან მუშაობა
array<int>^ masivi = gcnew array<int>(10);
for ( int indexi = 0; indexi < masivi->Length; indexi++ )
{
masivi[indexi] = 2 * ( indexi + 1 );
label1->Text += masivi[indexi].ToString() + "  ";
}
}
```

როგორც ვხედავთ, Length თვისებასთან მიმართვისთვის გამოიყენება -> ოპერატორი, რადგან masivi არის თვალყურის სადევნებელი დესკრიპტორი და მუშაობს როგორც მიმთითებელი. Length თვისება შეიცავს მასივის ელემენტების რაოდენობას (მასივის სიგრძეს), როგორც 32-თანრიგა მთელ რიცხვს. თუ გვინდა, რომ მასივის სიგრძე იყოს 64-თანრიგიანი მთელი რიცხვი, მაშინ უნდა გამოვიყენოთ LongLength თვისება.

CLR მასივის ინიციალიზება შეგვიძლია, აგრეთვე, gcnew ოპერატორის საშუალებით:

```
{
array<int>^ masivi = gcnew array<int> (5) { 10, 20, 30, 40, 50 };
for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        label1->Text += masivi[indexi].ToString() + "  ";
}
```

CLR მასივის ინიციალიზებისას, თუ გამოყოფილია უფრო დიდი მეხსიერება, ვიდრე მითითებული რიცხვების განლაგებას სჭირდება, მაშინ დარჩენილი მეხსიერება ნულებით შეივსება. მაგალითი:

```
{
array<int>^ masivi = gcnew array<int>(10) { 10, 20, 30, 40, 50 };
for each ( int cvladi in masivi )
        label1->Text += cvladi.ToString() + "  ";
}
```

ამ შემთხვევაში, მეხსიერებაში გამოიყოფა ადგილი masivi მასივის 10 ელემენტის მოსათავსებლად. აქედან, რიცხვებით შეივსება პირველი ხუთი რიცხვისთვის გამოყოფილი უბანი, დანარჩენი უბნები კი ნულებით შეივსება.

მასივის გამოცხადებისას მისი ზომა შეგვიძლია არ მივუთითოთ:

```
{
array<int>^ masivi = gcnew array<int> { 10, 20, 30, 40, 50 };
for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        label1->Text += masivi[indexi].ToString() + "  ";
}
```

ამ შემთხვევაში, masivi მასივი იკავებს ზუსტად იმ ზომის მეხსიერებას, რამდენი რიცხვიცაა მითითებული მისი ინიციალიზებისას, კერძოდ, 5 * 4 = 20 ბაიტს.

მასივის ცვლადი შეიძლება ინახავდეს ერთნაირი განზომილების მქონე ნებისმიერი მასივის მისამართს და ელემენტის ტიპს. მაგალითი:

```
array<int>^ masivi;
masivi = gcnew array<int>(10);
```

აქ იქმნება int ტიპის ახალი 10 ელემენტიანი ერთგანზომილებიანი მასივი, რომლის მისამართი masivi მასივში ინახება.

Object ტიპის ცვლადს ან ობიექტს შეგვიძლია მივანიჭოთ ნებისმიერი ტიპის ცვლადი ან ობიექტი. მოცემული პროგრამით Object ტიპის მასივი ივსება მთელი რიცხვებით.

```
//      Object ტიპის მასივის შევსება მთელი რიცხვებით
private: System::Void button8_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text  = "";
array<Object^>^  masivi = gcnew array<Object^> {1, 2, 3, 4, 5};
Int32 indexi, jami = 0;

for ( indexi = 0; indexi < masivi->Length; indexi++ )
{
        masivi[indexi] = indexi + 50;
        label1->Text += masivi[indexi]->ToString() + "  ";
}
for ( indexi = 0; indexi < masivi->Length; indexi++ )
jami += Convert::ToInt32(masivi[indexi]);
label2->Text = jami.ToString();
}
```

უნდა გვახსოვდეს, რომ CLR მასივის ელემენტები ობიექტებია.

მრავალგანზომილებიანი მასივი

შესაძლებელია ორი და მეტი განზომილების მქონე CLR მასივის შექმნა. მასივის მაქსიმალური განზომილებაა 32. ავტომატურად (ნაგულისხმევად) მასივის განზომილებაა 1. განზომილებას ვუთითებთ კუთხურ ფრჩხილებში ტიპის შემდეგ, მაგალითად

```
array<int, 2>^ masivi = gcnew array<int, 2>(3,4);
```

აქ იქმნება მთელრიცხვა ორგანზომილებიანი მასივი, რომელსაც აქვს 3 სტრიქონი და 4 სვეტი. სულ მასივი შედგება 3 x 4 = 12 ელემენტისგან. მოცემულ პროგრამაში იქმნება ორგანზომილებიანი მასივი, რომელიც მთელი რიცხვებით ივსება.

```
{
//      ორგანზომილებიანი მასივის შემთხვევითი რიცხვებით შევსება
label1->Text = "";
int striqonebis_raodenoba = Convert::ToInt32(textBox1->Text);
int svetebis_raodeboba = Convert::ToInt32(textBox2->Text);
```

```
array<int, 2>^ masivi = gcnew array<int, 2>(striqonebis_raodenoba, svetebis_raodeboba);

//	masivi მასივის შევსება შემთხვევითი რიცხვებით
for ( int striqoni = 0; striqoni < striqonebis_raodenoba; striqoni++ )
	for ( int sveti = 0; sveti < svetebis_raodeboba; sveti++ )
		masivi[striqoni, sveti] = striqoni + sveti;
//	masivi მასივის ეკრანზე გამოტანა
for ( int striqoni = 0; striqoni < striqonebis_raodenoba; striqoni++ )
{
	for ( int sveti = 0; sveti < svetebis_raodeboba; sveti++ )
		label1->Text += masivi[striqoni, sveti].ToString() + "  ";
	label1->Text += "\n";
}
}
```

მოცემული პროგრამით ხდება სამგანზომილებიან CLR მასივთან მუშობის დემონსტრირება.

```
//	სამგანზომილებიან მასივთან მუშაობა
private: System::Void button6_Click(System::Object^  sender, System::EventArgs^  e) {
array<int,3>^ masivi = gcnew array<int,3>(3,2,3);
int ind1, ind2, ind3, jami = 0;
//	მასივის შევსება რიცხვებით
for ( ind1 = 0; ind1 < 3; ind1++ )
	for ( ind2 = 0; ind2 < 2; ind2++ )
		for ( ind3 = 0; ind3 < 3; ind3++ )
			masivi[ind1, ind2, ind3] = ind1 + ind2 + ind3;
//	მასივის ელემენტების შეკრება
for ( ind1 = 0; ind1 < 3; ind1++ )
	for ( ind2 = 0; ind2 < 2; ind2++ )
		for ( ind3 = 0; ind3 < 3; ind3++ )
			jami += masivi[ind1, ind2, ind3];
label1->Text = jami.ToString();
}
```

მოცემულ პროგრამაში ხდება String^ ტიპის ერთ- და ორგანზომილებიან CLR მასივთან მუშობის დემონსტრირება.

```
//	String^ ტიპის ერთ- და ორგანზომილებიან მასივთან მუშაობა
private: System::Void button12_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = ""; label2->Text = "";
array<String^>^ masivi1 = gcnew array<String^>(5);
array<String^,2>^ masivi2 = gcnew array<String^,2>(2,2);
//	ერთგანზომილებიანი მასივის შევსება სტრიქონებით
for ( int indexi = 0; indexi < masivi1->Length; indexi++ )
	masivi1[indexi] = L"ინდექსი - " + indexi.ToString();
//	ერთგანზომილებიანი მასივის ელემენტების ეკრანზე გამოტანა
for ( int indexi = 0; indexi < masivi1->Length; indexi++ )
	label1->Text += masivi1[indexi] + "\n";
//	ორგანზომილებიანი მასივის შევსება სტრიქონებით
for ( int striqoni = 0; striqoni < 2; striqoni++ )
```

```
        for ( int sveti = 0; sveti < 2; sveti++ )
                masivi2[striqoni, sveti] = striqoni.ToString() + "," + sveti.ToString();
//      ორგანზომილებიანი მასივის ელემენტების ეკრანზე გამოტანა
for ( int striqoni = 0; striqoni < 2; striqoni++ )
{
        label2->Text += striqoni.ToString() + L"  სტრიქონის ინდექსები - " ;
        for ( int sveti = 0; sveti < 2; sveti++ )
                label2->Text += masivi2[striqoni, sveti] + "    ";
        label2->Text += "\n";
}
```

მოცემული პროგრამით ხდება Char ტიპის ერთგანზომილებიან CLR მასივთან მუშობის დემონსტრირება.

```
//      Char ტიპის ერთგანზომილებიან მასივთან მუშაობა
private: System::Void button13_Click(System::Object^  sender, System::EventArgs^  e) {
array<Char>^ masivi1 = gcnew array<Char> { L'ს', L'ა', L'ბ', L'ა' };
array<Char>^ masivi2 = gcnew array<Char>(6);
//      masivi2 მასივს ვავსებთ textBox1 კომპონენტში შეტანილი სტრიქონის სიმბოლოებით
for ( int indexi = 0; indexi < masivi2->Length; indexi++ )
{
        masivi2[indexi] = textBox1->Text[indexi];       //   textBox1 კომპონენტში შეგვაქვს 6 სიმბოლო
        label1->Text += masivi2[indexi].ToString();
}
for each ( Char simbolo in masivi1 )
        label2->Text += simbolo.ToString();
}
```

მასივების გამოცხადებისა და ინიციალიზების მაგალითები:

```
{
//      ერთგანზომილებიანი მასივების გამოცხადება
array<Int32>^    masivi11 = gcnew array<Int32>(4);
array<Double>^  masivi21 = gcnew array<Double>(7);
array<Boolean>^ masivi31 = gcnew array<Boolean>(5);
array<Char>^ masivi1 = gcnew array<Char>(5);
//      ორგანზომილებიანი მასივების გამოცხადება
array<String^,2>^ masivi8 = gcnew array<String^,2>(2,3);
array<Int32,3>^    masivi51 = gcnew array<Int32,3>(3,2,3);
array<Double,2>^  masivi61 = gcnew array<Double,2>(3,4);
array<Boolean,4>^ masivi71 = gcnew array<Boolean,4>(2,4,3,2);
array<Char,2>^ masivi2 = gcnew array<Char,2>(2,2);
//      მასივების ინიციალიზება
array<Object^>^  masivi9 = gcnew array<Object^>(5){1, 2, 3, 4, 5};
array<int>^ masivi10 = gcnew array<int> {1, 2, 3, 4, 5, 6};
array<Int32>^ masivi20 = gcnew array<Int32> {10, 20, 30, 40, 50, 60};
array<Double,2>^  masivi121 = gcnew array<Double,2>(3,4){ { 1.1, 2.2, 3.3, 4.4 }, { 3.3, 4.4, 5.5, 6.6 },
                                                          { 6.6, 7.7, 8.8, 9.9 } };
}
```

მასივების მასივი

მასივის ელემენტები შეიძლება იყოს ნებისმიერი ტიპის. ამიტომ, შეგვიძლია შევქმნათ მასივი, სადაც ელემენტები თვალყურის სადევნებელი დესკრიპტორებია, რომლებიც მასივებს მიმართავენ. ეს საშუალებას გვაძლევს შევქმნათ ე.წ. ***დაკბილული მასივი*** (jagged arrays), რადგან დესკრიპტორს შეუძლია მიმართოს მასივს, რომელსაც აქვს ელემენტების განსხვავებული რაოდენობა. მაგალითად, დავუშვათ, რომ პირველ კურსზე ოთხი ჯგუფია. თითოეულ ჯგუფში სტუდენტების განსხვავებული რაოდენობაა. მასივს ექნება სახე:

```
array<array<String^>^>^ masivi = gcnew array<array<String^>^>(4);
```

აქ masivi ცვლადი არის array<String^>^ ტიპის დესკრიპტორი. მასივში თითოეული ელემენტი, აგრეთვე არის მასივის დესკრიპტორი. შესაბამისად, მასივის ელემენტების ტიპს აქვს იგივე ფორმა - (array<type>^). ამიტომ, სწორედ ეს ეთითება საწყისი მასივის ტიპის განსაზღვრაში კუთხურ ფრჩხილებს შორის, რის შედეგად მიიღება array<array<type>^>^. მასივებში შენახული ელემენტები წარმოადგენენ String ტიპის ობიექტების დესკრიპტორებს. ამიტომ, მივიღებთ მასივის String^ ტიპს. ამრიგად, საბოლოოდ მივიღებთ array<array<String^>^>^ ტიპს.

ახლა შევქმნათ გვარების მასივები თითოეული ჯგუფისთვის:

```
masivi[0] = gcnew array<String^>
        { L"ქევხიშვილი", L"ჭუმბურიძე", L"სამხარაძე", L"კაპანაძე", L"ხოშტარია", L"კირვალიძე" };
masivi[1] = gcnew array<String^>  { L"ყალაბეგიშვილი", L"გაჩეჩილაძე", L"მჭედლიშვილი" };
masivi[2] = gcnew array<String^>  { L"მზარელუა", L"შენგელია", L"ფერაძე", L"მამაიაშვილი" };
masivi[3] = gcnew array<String^>  { L"ბარამიძე", L"ნონიაშვილი" };
```

შესაძლებელია მასივების მასივის ინიციალიზება მისი გამოცხადებისას:

```
//      მასივების მასივის ინიციალიზება მისი გამოცხადებას
{
array<array<String^>^>^ masivi = gcnew array<array<String^>^>
{
masivi[0] = gcnew array<String^>
        { L"ქევხიშვილი", L"ჭუმბურიძე", L"სამხარაძე", L"კაპანაძე", L"ხოშტარია", L"კირვალიძე" },
masivi[1] = gcnew array<String^>  { L"ყალაბეგიშვილი", L"გაჩეჩილაძე", L"მჭედლიშვილი" },
masivi[2] = gcnew array<String^>  { L"მზარელუა", L"შენგელია", L"ფერაძე", L"მამაიაშვილი" },
masivi[3] = gcnew array<String^>  { L"ბარამიძე", L"ნონიაშვილი" }
};
for ( int striqoni = 0; striqoni < masivi->Length; striqoni++ )
{
        for ( int sveti = 0; sveti < masivi[striqoni]->Length; sveti++ )
                label1->Text += masivi[striqoni][sveti] + "   ";
        label1->Text += "\n";
}
}
```

თვალყურის სადევნებელი მიმართვა

თვალყურის სადევნებელ მიმართვას აქვს C++ ენის მშობლიური მიმართვის მსგავსი შესაძლებლობები. იგი CLR გროვაში მოთავსებული ობიექტის ფსევდონიმია. შეგვიძლია ასეთი მიმართვა შევქმნათ სიდიდეების ტიპებზე სტეკში და მართვადი გროვის ობიექტების დესკრიპტორებზე. თვით თვალყურის სადევნებელი მიმართვა ყოველთვის იქმნება სტეკში და ავტომატურად გაახლდება თუ ობიექტი, რომელსაც იგი მიმართავს გადაადგილდება ნაგვის

შემგროვებლის მიერ.

თვალყურის სადევნებელი მიმართვა იქმნება % ოპერატორით:

```
{
int cvladi = Convert::ToInt32(textBox1->Text);
int% metvalkure_mimarTva = cvladi;

metvalkure_mimarTva += 100;
label1->Text = cvladi.ToString();
}
```

შიგა მიმთითებელი

არითმეტიკის ოპერაციის შესრულება არ შეიძლება თვალყურის სადევნებელი მიმთითებლის მისამართებზე. მიუხედავად ამისა, C++/CLI ითვალისწინებს მიმთითებლის ფორმას, რომლის მიმართ შეიძლება არითმეტიკის ოპერატორების გამოყენება. მას ***შიგა მიმთითებელი*** (interior pointer) ეწოდება და განისაზღვრება **interior_ptr** საკვანძო სიტყვის გამოყენებით. შიგა მიმთითებელში შენახული მისამართი შეიძლება ავტომატურად შეიცვალოს CLR ნაგვის შემგროვებლის მიერ, როდესაც ეს საჭიროა. შიგა მიმთითებელი ყოველთვის არის ავტომატური ცვლადი და ლოკალური ფუნქციის მიმართ. მაგალითი:

```
{
//      შიგა მიმთითებელთან მუშაობა
array<double>^ masivi = gcnew array<double> { 1.1, 2.2, 3.3, 4.4, 5.5 };
interior_ptr <double> shida_mimtitebeli = &masivi[0];

label1->Text = shida_mimtitebeli[2].ToString();
}
```

კუთხურ ფრჩხილებში მიეთითება იმ ობიექტის ტიპი, რომელსაც შიგა მიმთითებელი მიმართავს. შიგა მიმთითებელი შეიცავს მასივის ელემენტის მისამართს.

თუ შიგა მიმთითებლისათვის არ არის მითითებული საწყისი მონაცემები, მაშინ მისი ინიციალიზება ხდება nullptr-ით. მასივი ყოველთვისაა მოთავსებული CLR გროვაში. ამიტომ, ნაგვის შემგროვებელს შეუძლია შეცვალოს მისამართი, რომელიც მოთავსებულია შიგა მიმთითებელში.

შიგა მიმთითებლისათვის არსებობს შეზღუდვები ტიპის განსაზღვრაზე. იგი შეიძლება შეიცავდეს მნიშვნელობის კლასის ობიექტის მისამართს სტეკში ან ობიექტის დესკრიპტორის მისამართს CLR გროვაში. ის არ შეიძლება შეიცავდეს მთელი ობიექტის მისამართს CLR გროვაში. შიგა მიმთითებელი შეიძლება, აგრეთვე მიმართავდეს „მშობლიური“ კლასის ობიექტს ან „მშობლიურ“ მიმთითებელს.

შიგა მიმთითებელი შეგვიძლია გამოვიყენოთ მნიშვნელობის კლასის ობიექტის მისამართის შესანახად, რომელიც არის გროვაში მოთავსებული ობიექტის ნაწილი, როგორიცაა CLR მასივის ელემენტი. შეგვიძლია შევქმნათ შიგა მიმთითებელი, რომელსაც შეეძლება System::String ობიექტის დესკრიპტორის მისამართის შენახვა, მაგრამ ვერ შევქმნით შიგა მიმთითებელს თვით String ობიექტის მისამართის შესანახად. მაგალითი.

```
interior_ptr <String^> striqoni1;
interior_ptr <String> striqoni2;                    //      შეცდომა
```

შიგა მიმთითებლის მიმართ შეიძლება გამოვიყენოთ არითმეტიკის ის ოპერაციები, რომლებსაც ვიყენებთ C++ ენის მშობლიური მიმთითებლის მიმართ:

```
{
```

```
//      შიგა მიმთითებლების მიმართ არითმეტიკის ოპერაციების გამოყენება
array<int>^ masivi = gcnew array<int> { 1, 2, 3, 4, 5 };
interior_ptr<int> sawyisi_misamarti = &masivi[0];
interior_ptr<int> bolo_misamarti = &masivi[masivi->Length - 1];
int jami = 0;
//      გამოითვლება მასივის ელემენტების ჯამი
for ( ; sawyisi_misamarti <= bolo_misamarti; )
jami += *sawyisi_misamarti++;
label1->Text = jami.ToString();
array<String^>^ striqonebi = gcnew array<String^> {L"ანა", L"საბა", L"ლიკა"};
for ( interior_ptr<String^> striqonis_misamarti = &striqonebi[0]; striqonis_misamarti-&striqonebi[0] <
striqonebi->Length; ++striqonis_misamarti )
        label2->Text += *striqonis_misamarti + "   ";
}
```

for each ციკლი

for each ციკლი გამოიყენება კოლექციის ელემენტების დასამუშავებლად. **კოლექცია** არის ობიექტების სიმრავლე. C++ ენაში განსაზღვრულია კოლექციების რამდენიმე ტიპი, ერთ-ერთი მათგანია მასივი. for each ციკლის სინტაქსია:

for each (***ტიპი ცვლადის_სახელი*** **in** ***კოლექცია*** **)**
{
ციკლის ტანი
}

აქ ***ტიპი*** არის იტერაციული ცვლადის ტიპი და უნდა ემთხვეოდეს კოლექციის ტიპს. ***ცვლადის_სახელი*** არის იტერაციული ცვლადის სახელი. იტერაციული ცვლადი რიგ-რიგობით იღებს კოლექციის ელემენტების მნიშვნელობებს. კოლექცია შეიძლება იყოს მასივი, სტრიქონი და ა.შ. იტერაციული ცვლადი მისაწვდომია მხოლოდ წაკითხვის რეჟიმში, ამიტომ, ვერ შევძლებთ მასივის შეცვლას იტერაციული ცვლადისთვის ახალი მნიშვნელობის მინიჭების გზით.

ქვემოთ მოცემულია for each ციკლის მაგალითი. თავიდან იქმნება მთელრიცხვა მასივი და სრულდება მისი ინიციალიზება. შემდეგ ეს მნიშვნელობები გამოიცემა label კომპონენტში და გამოითვლება მათი ჯამი.

```
{
//      მასივის ელემენტების ჯამის გამოთვლა for each ციკლის გამოყენებით
label1->Text =""; label2->Text ="";
array<int>^ masivi = gcnew array<int> {1, 2, 3, 4, 5};
int jami = 0;

for each ( int cvladi in masivi )
jami += cvladi;
        label1->Text = jami.ToString();

for each ( int cvladi in masivi )
        label2->Text += cvladi.ToString() + "   ";
}
```

როგორც ვხედავთ მასივის ელემენტების დამუშავება ხდება მიმდევრობით, დაწყებული პირველი ელემენტიდან, ინდექსის გამოყენების გარეშე.

for each ციკლის საშუალებით მასივი შეიძლება დავამუშავოთ მხოლოდ დასაწყისიდან ბოლოსკენ. ქვემოთ მოცემულია მასივის მაქსიმალური და მინიმალური ელემენტების პოვნის პროგრამა for each ციკლის გამოყენებით.

```
{
//      მასივის მაქსიმალური და მინიმალური ელემენტების პოვნა
int max, min;
array<int>^ masivi = gcnew array<int>(10) {9, -1, 101, 8, -9, 56, 3, -2, 87, 81};

label1->Text = "";
max = min = masivi[0];
//      მასივის ელემენტების ეკრანზე გამოტანა
for each ( int ricxvi in masivi )
        label1->Text += ricxvi.ToString() + " ";
//      მინიმალური და მაქსიმალური ელემენტების პოვნა
for each ( int ricxvi in masivi )
{
        if ( ricxvi > max ) max = ricxvi;
        if ( ricxvi < min ) min = ricxvi;
}
label2->Text = max.ToString();
label3->Text = min.ToString();
}
```

ქვემოთ მოცემულ პროგრამაში for each ციკლი გამოიყენება ორგანზომილებიანი მასივის ელემენტების ჯამის გამოსათვლელად.

```
{
//      ორგანზომილებიანი მასივის ელემენტების ჯამის გამოთვლა
int max, min;
array<int,2>^ masivi = gcnew array<int,2>(3,3) {
                                                { 1, 2, -3 },
                                                { -4, 5, 6 },
                                                { 7, -8, 9 },
                                                  };
label1->Text = "";
int jami = 0;

for each ( int ricxvi in masivi )
{
        label1->Text += ricxvi.ToString() + "   ";
        jami += ricxvi;
}
label2->Text = jami.ToString();
}
```

მოცემულ პროგრამაში for each ციკლის გამოიყენებით სტრიქონში ხდება ხმოვნებისა და თანხმოვნების რაოდენობის დათვლა.

```
{
```

```
//        სტრიქონში ხმოვნებისა და თანხმოვნების რაოდენობის დათვლა
String^ striqoni = L"რომანსამხარაძე";
int xmovnebis_raodenoba = 0;
int tanxmovnebis_raodenoba = 0;

for each ( Char simbolo in striqoni )
switch ( simbolo )
{
        case L'ა' : case L'ი' : case L'ო' : case L'ე' : case L'უ' : xmovnebis_raodenoba++; break;
        default : tanxmovnebis_raodenoba++; break;
}
label1->Text = tanxmovnebis_raodenoba.ToString();
label2->Text = xmovnebis_raodenoba.ToString();
}
```

ბიტობრივი ოპერატორი

ბიტობრივი (თანრიგობრივი) ოპერატორის ოპერანდები შეიძლება იყოს მთელი რიცხვები, აგრეთვე, bool ტიპის მნიშვნელობები. ეს ოპერატორები არ შეიძლება გამოვიყენოთ, float ან double ტიპის მნიშვნელობების, აგრეთვე, კლასის ობიექტების მიმართ. ასეთ ოპერატორებს ბიტობრივი ეწოდება იმიტომ, რომ გამოიყენებიან მთელი რიცხვების ბიტების შესამოწმებლად, დასაყენებლად (1 მდგომარეობაში გადაყვანა) და ძვრისათვის. ბიტობრივი ოპერაციები ხშირად გამოიყენება სისტემურ დონეზე დაპროგრამებისათვის, მაგალითად, როდესაც საჭიროა ინფორმაციის მიღება მოწყობილობის მდგომარეობის შესახებ და ა.შ. ბიტობრივი ოპერაციები მოცემულია 4.1 ცხრილში.

ცხრილი 4.1. ბიტობრივი ოპერაციები

ოპერატორი	აღწერა
&	ბიტობრივი ოპერატორი AND
|	ბიტობრივი ოპერატორი OR
^	ბიტობრივი ოპერატორი XOR
>>	მარჯვნივ ძვრის ოპერატორი
<<	მარცხნივ ძვრის ოპერატორი
~	უნარული ოპერატორი NOT

&, |, ^ და ~ ბიტობრივი ოპერატორები

&, |, ^ და ~ ბიტობრივი ოპერატორები ისეთივე ოპერაციებს ასრულებენ, როგორსაც მათი ბულის ეკვივალენტები. განსხვავება ისაა, რომ ბიტობრივი ოპერატორები ოპერანდებზე ზემოქმედებენ ბიტების დონეზე. 4.2 ცხრილში მოცემულია თითოეული ოპერაციის შესრულების შედეგები.

ცხრილი 4.2. ბიტობრივი ოპერაციები

b1	b2	b1&b2	b1 | b2	b1 ^ b2	~b1
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

ბიტობრივი & ოპერაცია შეგვიძლია გამოვიყენოთ როგორც ბიტის განულების (ჩამოგდების) საშუალება. ეს იმას ნიშნავს, რომ თუ რომელიმე ოპერანდის ერთ-ერთ ბიტს აქვს მნიშვნელობა 0, მაშინ შედეგის შესაბამის ბიტსაც ექნება მნიშვნელობა 0. მაგალითად,

```
11010011
      &
10101010
--------
10000010
```

მოცემულ პროგრამით ნაჩვენებია & ოპერატორის გამოყენების მაგალითი. პროგრამა ქვედა რეგისტრის სიმბოლოებს გარდაქმნის ზედა რეგისტრის სიმბოლოებად მეექვსე ბიტის განულების გზით. Unicode/ASCII სიმბოლოების სტანდარტულ ნაკრებში ქვედა რეგისტრის სიმბოლოების მნიშვნელობა მეტია ზედა რეგისტრის სიმბოლოების შესაბამის მნიშვნელობაზე 32 ერთეულით ($32_{10}=00100000_2$). შედეგად, ქვედა რეგისტრის სიმბოლოების გადასაყვანად ზედა რეგისტრში საკმარისია მეექვსე ბიტის განულება. მაგალითი:

```
{
//      ქვედა რეგისტრის სიმბოლოების გარდაქმნა ზედა რეგისტრის სიმბოლოებად
Char simbolo;
label1->Text = "";

for ( int indexi = 0; indexi < 10; indexi++ )
  {
        simbolo = (char) ( 'a' + indexi );
        label1->Text += simbolo + " - ";
        simbolo = (char) ( simbolo & 65503 );
        label1->Text += simbolo + "\n";
  }
}
```

რიცხვი 65503 ორბაიტიანია და ორობით სისტემაში ასე გამოისახება 1111 1111 1101 1111. & ოპერაცია უცვლელად ტოვებს simbolo ცვლადის ყველა ბიტს მეექვსე ბიტის გარდა, რომელიც განულდება. & ოპერაციის გამოყენება შეგვიძლია მაშინ, როდესაც საჭიროა განისაზღვროს ბიტი დაყენებულია (1) თუ ჩამოგდებული (0). მაგალითად, ქვემოთ მოცემული if ოპერატორით მოწმდება status ცვლადის მეოთხე ბიტის მნიშვნელობა:

```
if ( status & 8 == true ) label2->Text = L"მეოთხე ბიტი დაყენებულია";
```

რიცხვი 8 გამოიყენება იმიტომ, რომ მის ორობით წარმოდგენაში დაყენებულია მხოლოდ მეოთხე ბიტი (0000 1000). შედეგად, if ოპერატორი true შედეგს მხოლოდ მაშინ გასცემს, როდესაც status ცვლადის მნიშვნელობაში მეოთხე ბიტი იქნება დაყენებული. ქვემოთ მოცემული პროგრამით ასეთი მიდგომა გამოიყენება ეკრანზე Byte ტიპის მნიშვნელობის ორობითი წარმოდგენის გამოსატანად.

```
{
//      ათობითი რიცხვის გარდაქმნა ორობით რიცხვად
```

```
int indeqsi;
Byte ricxvi = Convert::ToByte(textBox1->Text);

label1->Text = " ";
for ( indeqsi = 128; indeqsi > 0; indeqsi /= 2 )
  {
        if ( ( ricxvi & indeqsi ) != 0 ) label1->Text += "1";
              else label1->Text += "0";
  }
}
```

for ციკლში & ოპერატორის გამოყენებით მოწმდება ricxvi ცვლადის თითოეული ბიტი. თუ ის დაყენებულია, მაშინ label2 კომპონენტში გამოჩნდება 1, თუ ჩამოგდებულია - 0.

& ოპერატორის საწინააღმდეგოდ | ოპერატორი შეგვიძლია გამოვიყენოთ ბიტების დასაყენებლად. თუ ერთ-ერთ ოპერანდში ბიტი დაყენებულია, მაშინ შედეგში შესაბამისი ბიტი იქნება დაყენებული. მაგალითად,

11010011
|
10101010
11111011

ზედა რეგისტრის სიმბოლოების გარდასაქმნელად ქვედა რეგისტრის სიმბოლოებად გამოვიყენოთ | ოპერატორი, როგორც ეს ქვემოთაა ნაჩვენები:

```
{
//      ზედა რეგისტრის სიმბოლოების გარდაქმნა ქვედა რეგისტრის სიმბოლოებად
Char simbolo;

for ( int indeqsi = 0; indeqsi < 10; indeqsi++ )
  {
        simbolo = (Char) ( 'A' + indeqsi );
        label1->Text += simbolo + " - ";
//      ამ ოპერატორის შესრულება აყენებს მეექვსე ბიტს
//      შედეგად simbolo ცვლადში ჩაიწერება ქვედა რეგისტრის სიმბოლო
        simbolo = (Char) ( simbolo | 32 );
        label1->Text += simbolo + "\n";
  }
}
```

| ოპერაცია გამოიყენება თითოეული სიმბოლოს მნიშვნელობისა და 32 მნიშვნელობის მიმართ, რომელიც ორობით სისტემაშ ასე გამოისახება 0000 0000 0010 0000. ამ ორობით წარმოდგენაში დაყენებულია მხოლოდ მეექვსე ბიტი. თუ | ოპერაციის ერთი ოპერანდია რიცხვი 32, მეორე ოპერანდი კი - ნებისმიერი რიცხვი, მაშინ შედეგად მივიღებთ მნიშვნელობას, რომელშიც დაყენებული იქნება მეექვსე ბიტი. დანარჩენი ბიტები დარჩება უცვლელი.

^ ოპერაციის შესრულების შედეგად ბიტის დაყენება მოხდება მხოლოდ მაშინ, როდესაც ორივე ბიტი განსხვავებულია. მაგალითად,

11010011
^
10101010
11111011

^ ოპერატორს საკმაოდ საინტერესო თვისება აქვს, რომელიც საშუალებას გვაძლევს ის გამოვიყენოთ შეტყობინების ან რაიმე ინფორმაციის დასაშიფრად. თუ ამ ოპერატორს გამოვიყენებთ x და y ცვლადების მიმართ და შემდეგ მიღებული შედეგისა და y ცვლადის მიმართ, მივიღებთ x მნიშვნელობას, ე.ი.

```
R1 = x ^ y;
R2 = R1 ^ y;
```

ოპერატორების შესრულების შედეგად R2 ცვლადს მიენიჭება x ცვლადის მნიშვნელობა. ქვემოთ მოცემულ პროგრამაში ეს პრინციპია გამოყენებული შეტყობინების დაშიფვრისა და გაშიფვრისათვის. პირველად ^ ოპერატორი გამოიყენება დაშიფვრისათვის, მეორედ კი - გაშიფვრისათვის. შედეგად, მივიღებთ საწყის ტექსტს. დაშიფვრისას რაიმე მთელი რიცხვი შეგვიძლია გამოვიყენოთ როგორც გასაღები. ქვემოთ მოცემული პროგრამით ხდება დაშიფვრისა და გაშიფვრის დემონსტრირება.

```
{
//      დაშიფვრისა და გაშიფვრის დემონსტრირება
String^ teqsti = L"შეტყობინების დაშიფვრა";
String^ dashifruli_teqsti = "";
String^ gashifruli_teqsti = "";
int key = 55;

label1->Text = L"დასაშიფრი შეტყობინება:   " + teqsti;
//      შეტყობინების დაშიფვრა
for ( int indeqsi = 0; indeqsi < teqsti->Length; indeqsi++ )
        dashifruli_teqsti += (Char) ( teqsti[indeqsi] ^ key );
label2->Text = L" დაშიფრული შეტყობინება:   " + dashifruli_teqsti;
//      შეტყობინების გაშიფვრა
for ( int indeqsi = 0; indeqsi < teqsti->Length; indeqsi++ )
        gashifruli_teqsti += (Char) ( dashifruli_teqsti[indeqsi] ^ key );
label3->Text = L"გაშიფრული შეტყობინება:   " + gashifruli_teqsti;
}
```

უნარული ~ ოპერაცია ცვლის ოპერანდის ყველა ბიტის მნიშვნელობას საწინააღმდეგოთი. მაგალითად, თუ A = 1001 0011, მაშინ ~A ოპერაციის შედეგი იქნება 0110 1100.

ქვემოთ მოცემულ პროგრამაში ხდება ~ ოპერატორის გამოყენების დემონსტრირება. ეკრანზე ჯერ ჩნდება რიცხვი 35-ის (0010 0011) ორობითი წარმოდგენა, შემდეგ კი ამ რიცხვის მიმართ ~ ოპერაციის გამოყენების შედეგი (1101 1100).

```
{
//      ~ ოპერატორის მუშაობის დემონსტრირება
Byte ricxvi = 35;

label1->Text = ""; label2->Text = "";
//      ricxvi ცვლადის ორობითი წარმოდგენის label კომპონენტში გამოტანა
for ( int indeqsi = 128; indeqsi > 0; indeqsi /= 2 )
        if ( ( ricxvi & indeqsi ) != 0 ) label1->Text += " 1";
                else  label1->Text += " 0";
//      ricxvi ცვლადის ორობითი წარმოდგენის ინვერსია და label კომპონენტში გამოტანა
ricxvi = (Byte) ~ricxvi;
for ( int indeqsi = 128; indeqsi > 0; indeqsi /= 2 )
```

```
        if ( ( ricxvi & indeqsi ) != 0 ) label2->Text += " 1";
                else  label2->Text += " 0";
}
```

ყველა ორობითი ბიტობრივი ოპერატორი შეგვიძლია ჩავწეროთ შედგენილი ბიტობრივი ოპერატორების სახით. მაგალითად,

```
x ^= 127;
y &= 63;
z |= 31;
```

ძვრის ოპერატორები

არსებობს ოპერანდების მარცხნივ და მარჯვნივ ძვრის ოპერატორები. მათი სინტაქსია:

ცვლადის_სახელი << ბიტების_რაოდენობა

ცვლადის_სახელი >> ბიტების_რაოდენობა

მარცხნივ ძვრა (<<) იწვევს ყველა ბიტის გადაადგილებას ერთი ბიტით მარცხნივ, ამასთან, თავისუფლდება უმცროსი ბიტები, რომლებიც ნულებით ივსება, ხოლო უფროსი ბიტების შესაბამისი რაოდენობა იკარგება. მარჯვნივ ძვრა (>>) იწვევს ყველა ბიტის ერთი ბიტით მარჯვნივ გადაადგილებას, ამასთან, თუ მარჯვნივ იძვრის უნიშნო მნიშვნელობის ბიტები, მაშინ უფროსი თანრიგები ნულებით შეივსება და უმცროსი თანრიგების შესაბამისი რაოდენობა იკარგება. ნიშნიანი მნიშვნელობის მარჯვნივ ძვრისას ხდება ნიშნის ბიტის შენახვა. ნიშნიანი დადებითი რიცხვისთვის უფროსი ბიტი არის 0, უარყოფითი რიცხვისთვის - 1.

ქვემოთ მოცემული პროგრამით ნაჩვენებია ძვრის ოპერაციების გამოყენების მაგალითი.

```
{
//      ძვრის ოპერატორებთან მუშაობის დემონსტრირება
int ricxvi = 1;

label1->Text = ""; label2->Text = "";
//      ricxvi ცვლადის ორობითი წარმოდგენის label კომპონენტში გამოტანა
for ( int i = 0; i < 8; i++ )
  {
        for ( int j = 128; j > 0; j /= 2 )
                if ( ( ricxvi & j ) != 0 ) label1->Text += "1";
                        else label1->Text += "0";
        label1->Text += "\n";
        ricxvi = ricxvi << 1;               //      ძვრა მარცხნივ ერთი ბიტით
  }
ricxvi = 128;
//      ricxvi ცვლადის ორობითი წარმოდგენის label კომპონენტში გამოტანა
for ( int i = 0; i < 8; i++ )
  {
        for ( int j = 128; j > 0; j /= 2 )
                if ( ( ricxvi & j ) != 0 ) label2->Text += "1";
                        else label2->Text += "0";
        label2->Text += "\n";
ricxvi = ricxvi >> 1;                       //      ძვრა მარჯვნივ ერთი ბიტით
  }
}
```

თავი 5. ფუნქციები

ფუნქცია

ფუნქცია არის კოდის თვითსაკმარისი ნაწილი, რომელიც გარკვეულ ამოცანას ასრულებს. მას აქვს სახელი და შეიძლება პროგრამული კოდის ერთ ან მეტ სტრიქონს შეიცავდეს. ფუნქციის სახელად შეგვიძლია ნებისმიერი იდენტიფიკატორის გამოყენება. საკვანძო სიტყვის გამოყენება ფუნქციის სახელად არ შეიძლება. Main() სახელი დარეზერვებულია იმ ფუნქციისთვის, საიდანაც პროგრამის შესრულება იწყება. ძირითად ფუნქციაში ფუნქციის გამოძახება ბევრჯერ შეიძლება.

ფუნქცია შედგება სათაურისა და კოდისაგან (ტანისაგან). ფუნქციის სათაური შედგება ფუნქციის მიერ გაცემული შედეგის ტიპის, ფუნქციის სახელისა და პარამეტრების სიისაგან.

ფუნქციის გამოცხადების სინტაქსია:

შედეგის_ტიპი ფუნქციის_სახელი(პარამეტრების_სია)
{
ფუნქციის კოდი
}

აქ, ***შედეგის_ტიპი*** არის ფუნქციის მიერ გასაცემი მნიშვნელობის ტიპი. თუ ფუნქცია მნიშვნელობას არ გასცემს, მაშინ უნდა მივუთითოთ void ტიპი. ***ფუნქციის_სახელი*** არის ნებისმიერი იდენტიფიკატორი. ***პარამეტრების_სია*** შედგება პარამეტრების სახელებისაგან, რომლებიც ერთმანეთისაგან მძიმეებით გამოიყოფა. თითოეული პარამეტრის სახელის წინ უნდა ეწეროს შესაბამისი ტიპი. პარამეტრი ესაა ცვლადი, რომელიც იღებს არგუმენტის მნიშვნელობებს, რომლებიც ფუნქციას გადაეცემა მისი გამოძახებისას. ფუნქციას პარამეტრები შეიძლება არ ჰქონდეს.

არ შეიძლება ერთი ფუნქციის შიგნით მეორე ფუნქციის განსაზღვრა. არ არის აუცილებელი გლობალური ცვლადების ჩართვა პარამეტრების სიაში, რადგან ისინი ყოველთვის მისაწვდომია ფუნქციისათვის.

ფუნქცია უნდა მოვათავსოთ

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
```

სტრიქონის თავზე

```
#pragma endregion
```

სტრიქონის შემდეგ.

შევადგინოთ ფუნქცია, რომელიც ეკრანზე გამოიტანს რაიმე ტექსტს:

```
void Function1(){
	MessageBox::Show(L"C++ დაპროგრამების ენაა");
}
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
	Function1();
}
```

ამ შემთხვევაში, Function1() ფუნქცია არაფერს არ გვიბრუნებს, ამიტომ მისი სახელის მარცხნივ მითითებულია void სიტყვა. ამ ფუნქციას პარამეტრების არ აქვს. მის გამოსაძახებლად ძირითად პროგრამაში ვუთითებთ მხოლოდ ფუნქციის სახელს.

პარამეტრის გამოყენება

გამოძახებისას ფუნქციას შეგვიძლია გადავცეთ ერთი ან მეტი მნიშვნელობა.

ფუნქციისთვის გადასაცემ მნიშვნელობას არგუმენტი ეწოდება. ფუნქციის პარამეტრების სიაში გამოცხადებულ ცვლადს, რომელიც იღებს არგუმენტის მნიშვნელობას, პარამეტრი ეწოდება. პარამეტრების გამოცხადება ხდება მრგვალი ფრჩხილების შიგნით, რომლებიც ფუნქციის სახელს მოსდევს. პარამეტრის გამოცხადების სინტაქსი ცვლადების გამოცხადების სინტაქსის ანალოგიურია. პარამეტრი იმყოფება თავისი ფუნქციის ხილვადობის უბანში, ანუ ხილულია მხოლოდ ამ ფუნქციის შიგნით და მოქმედებს როგორც ლოკალური ცვლადი. თუ ფუნქციას რამდენიმე პარამეტრი აქვს, მაშინ ისინი ერთმანეთისაგან მძიმეებით გამოიყოფა.

ახლა შევადგინოთ ფუნქცია, რომელიც label1 კომპონენტში გამოგვიტანს რაიმე ტექსტს:

```
#pragma endregion
void Function1(Label^ lab1){
	label1->Text = L"C++ დაპროგრამების ენაა";
}
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
	Function1(label1);
}
```

Function1() ფუნქცია ამ შემთხვევაშიც არაფერს არ გვიბრუნებს, მაგრამ მას გადავეცით ერთი პარამეტრი, კერძოდ Label ტიპის არგუმენტი. ძირითად პროგრამაში ამ ფუნქციის გამოძახებისას lab1 პარამეტრს გადაეცემა label1 კომპონენტი, რომელშიც გამოჩნდება საჭირო ტექსტი.

ფუნქციიდან მართვის დაბრუნება

ფუნქციიდან მართვის დაბრუნება ანუ ფუნქციის შესრულების შეწყვეტა და გამომძახებელი პროგრამისათვის მართვის დაბრუნება, ორ შემთხვევაში ხდება. პირველი, როდესაც გვხვდება ფუნქციის დამხურავი ფიგურული ფრჩხილი და მეორე, როდესაც სრულდება return ოპერატორი. არსებობს return ოპერატორის ორი ფორმა. პირველი ფორმა გამოიყენება ფუნქციაში, რომელსაც void ტიპი აქვს, ანუ მნიშვნელობას არ აბრუნებს, მეორე კი - ფუნქციაში, რომელიც მნიშვნელობას აბრუნებს.

ფუნქციაში, რომელიც მნიშვნელობას არ აბრუნებს, გამოიყენება შემდეგი სინტაქსის return ოპერატორი:

[return;]

ასეთ ფუნქციაში return ოპერატორი შეგვიძლია არც მივუთითოთ.

return ოპერატორის შესრულებისას მართვა გადაეცემა გამომძახებელი პროგრამის იმ ადგილს, საიდანაც ფუნქცია იყო გამოძახებული, ამასთან ფუნქციის კოდის დარჩენილი ნაწილი არ შესრულდება.

ფუნქციის ერთ-ერთი მნიშვნელოვანი თვისებაა მის მიერ მნიშვნელობის დაბრუნება (გაცემა). ფუნქციები მნიშვნელობას უბრუნებენ გამომძახებელ პროგრამას შემდეგი სინტაქსის მქონე return ოპერატორის გამოყენებით:

return *მნიშვნელობა;*

შევადგინოთ ფუნქცია, რომელიც გამოთვლის და დაგვიბრუნებს სამკუთხედის პერიმეტრს.

```
int Perimetri(int gv1, int gv2, int gv3){
	return gv1 + gv2 + gv3;
}
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
//	პირველი სამკუთხედის გვერდები
```

```
        int gverdi11, gverdi12, gverdi13, perimetri1;
//      მეორე სამკუთხედის გვერდები
        int gverdi21, gverdi22, gverdi23, perimetri2;
//
        gverdi11 = int::Parse(textBox1->Text);
        gverdi12 = int::Parse(textBox2->Text);
        gverdi13 = int::Parse(textBox3->Text);
//
        gverdi21 = int::Parse(textBox4->Text);
        gverdi22 = int::Parse(textBox5->Text);
        gverdi23 = int::Parse(textBox6->Text);
//      ფუნქციის გამოძახება პირველი სამკუთხედის პერიმეტრის გამოსათვლელად
        perimetri1 = Perimetri(gverdi11, gverdi12, gverdi13);
//      ფუნქციის გამოძახება მეორე სამკუთხედის პერიმეტრის გამოსათვლელად
        perimetri2 = Perimetri(gverdi21, gverdi22, gverdi23);
//
        label1->Text = perimetri1.ToString();
        label2->Text = perimetri2.ToString();
}
```

აქ გამოცხადებულია Perimetri() ფუნქცია, რომელიც გვიბრუნებს (გასცემს) მთელ რიცხვს და აქვს სამი მთელი ტიპის პარამეტრი. ძირითად პროგრამაში ეს ფუნქცია ორჯერ გამოიძახება

```
perimetri1 = Perimetri(gverdi11, gverdi12, gverdi13);
perimetri2 = Perimetri(gverdi21, gverdi22, gverdi23);
```

სტრიქონებში. პირველი გამოძახების შემთხვევაში, ფუნქციის პარამეტრებს გადაეცემათ gverdi11, gverdi12 და gverdi13 არგუმენტები. კერძოდ, gv1 პარამეტრს მიენიჭება gverdi11 არგუმენტი, gv2 პარამეტრს - gverdi12 არგუმენტი, ხოლო gv3 პარამეტრს - gverdi13 არგუმენტი. ფუნქცია დაგვიბრუნებს შესაბამისი მნიშვნელობის მქონე პერიმეტრს, რომელიც მიენიჭება perimetri1 ცვლადს. მეორე გამოძახების შემთხვევაში, ფუნქციის პარამეტრებს გადაეცემათ gverdi21, gverdi22 და gverdi23 არგუმენტები. კერძოდ, gv1 პარამეტრს მიენიჭება gverdi21 არგუმენტი, gv2 პარამეტრს - gverdi22 არგუმენტი, ხოლო gv3 პარამეტრს - gverdi23 არგუმენტი. ფუნქცია დაგვიბრუნებს შესაბამისი მნიშვნელობის მქონე პერიმეტრს, რომელიც მიენიჭება perimetri2 ცვლადს.

ქვემოთ მოყვანილია პროგრამა, რომელშიც ფუნქციისათვის მნიშვნელობის გადასაცემად პარამეტრი გამოიყენება. ArisLuwi კლასის შიგნით განსაზღვრული luwi_kenti(int par1) ფუნქცია აბრუნებს bool ტიპის შედეგს. თუ მისთვის გადაცემული მნიშვნელობა არის ლუწი, მაშინ ის გასცემს true მნიშვნელობას, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას.

```
//      ფუნქციის განსაზღვრა, რომელიც ამოწმებს რიცხვი კენტია თუ ლუწი
bool Luwi_kenti( int par1) {
if ( (par1 % 2 ) == 0 ) return true;
        else return false;
}
//
private: System::Void button6_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi;

ricxvi = Convert::ToInt32(textBox1->Text);
if ( Luwi_kenti(ricxvi) ) label1->Text = L" რიცხვი " + ricxvi.ToString() + L" ლუწია";
        else  label1->Text = L" რიცხვი " + ricxvi.ToString() + L" კენტია";
```

```
}
```

Luwi_kenti() ფუნქციის გამოძახებისას ricxvi არგუმენტის მნიშვნელობა მიენიჭება par1 პარამეტრს.

ახლა შევადგინოთ arisJeradi() ფუნქცია, რომელიც განსაზღვრავს არის თუ არა პირველი პარამეტრი მეორე პარამეტრის ჯერადი.

```
//      ფუნქცია ამოწმებს  პირველი პარამეტრი უნაშთოდ იყოფა თუ არა მეორე პარამეტრზე
bool arisJeradi(int par1, int par2) {
if ( ( par1 % par2 ) == 0 ) return true;
        else return false;
}
//
private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
//      ფუნქციის გამოძახება
if ( arisJeradi(ricxvi1, ricxvi2) )

label2->Text = ricxvi1.ToString() + L" არის " + ricxvi2.ToString() + L"-ის ჯერადი";
else    label2->Text = ricxvi1.ToString() + L" არ არის " + ricxvi2.ToString() + L"-ის ჯერადი";
}
```

arisJeradi() ფუნქციის გამოძახებისას ricxvi1 არგუმენტის მნიშვნელობა მიენიჭება par1 პარამეტრს, ricxvi2 არგუმენტის მნიშვნელობა კი - par2 პარამეტრს.

ფუნქციისთვის მშობლიური და CLR მასივების გადაცემა

ფუნქციას შეგვიძლია გადავცეთ როგორ მშობლიური, ისე CLR მასივი. ორივე შემთხვევაში, ფუნქციას გადაეცემა მასივი საწყისი მისამართი, ანუ მასივის პირველი ელემენტის მისამართი.

ქვემოთ მოცემულ პროგრამაში Minimumi() ფუნქციას გადაეცემა ერთგანზომილებიანი მშობლიური მასივი. ფუნქცია პოულობს და აბრუნებს მასივის მინიმალური ელემენტის მნიშვნელობას.

```
//      ფუნქციისთვის ერთგანზომილებიანი მშობლიური მასივის გადაცემის დემონსტრირება
int Minimumi(int mas[]) {
int min_ricxvi = mas[0];
for ( int indexi =1; indexi < 5; indexi++ )
if  (min_ricxvi > mas[indexi] ) min_ricxvi = mas[indexi];
return min_ricxvi;
}
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
label2->Text = "";
int masivi[5] = { 5, -1, 8, -4, 2 };

int shedegi = Minimumi(masivi);                 //      Minimumi ფუნქციას masivi მასივი გადაეცემა
        label1->Text = shedegi.ToString();
for ( int indexi = 0; indexi < 5; indexi++ )
```

```
        label2->Text += masivi[indexi].ToString() + "  ";
}
```

აქ, პროგრამის int shedegi = Minimumi(masivi); სტრიქონში ხდება Minimumi() ფუნქციის გამოძახება. მას გადაეცემა masivi მასივის საწყისი მისამართი (პირველი ელემენტის მისამართი), რომელიც მიენიჭება mas პარამეტრს. შედეგად, masivi და mas მასივები მეხსიერების ერთსა და იმავე უბანს მიმართავენ. ამიტომ, მასივთან მუშაობა შესრულდება იმ უბანში, რომელიც masivi მასივისთვის გამოიყო.

ქვემოთ მოცემულ პროგრამაში Minimumi() ფუნქციას გადაეცემა ორგანზომილებიანი მშობლიური მასივი. ფუნქცია პოულობს და აბრუნებს მასივის მინიმალური ელემენტის მნიშვნელობას. ამ შემთხვევაშიც, ფუნქციას გადაეცემა მისი პირველი ელემენტის მისამართი.

```
//      ფუნქციისთვის ორგანზომილებიანი მშობლიური მასივის გადაცემის დემონსტრირება
int Minimumi(int mas[][3]) {
int min_ricxvi = mas[0][0];
for ( int striqoni =0; striqoni < 3; striqoni++ )
        for ( int sveti =0; sveti < 3; sveti++ )
if  (min_ricxvi > mas[striqoni][sveti] ) min_ricxvi = mas[striqoni][sveti];
return min_ricxvi;
}
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
label2->Text = "";
int masivi[3][3] = { { 5, -1, 8}, {-4, 2, 3}, { 2, 7, -3} };

int shedegi = Minimumi(masivi);                 //      Minimumi ფუნქციას masivi მასივი გადაეცემა
label1->Text = shedegi.ToString();
for ( int striqoni =0; striqoni < 3; striqoni++ )
{
for ( int sveti =0; sveti < 3; sveti++ )
        label2->Text += masivi[striqoni][sveti].ToString() + "  ";
label2->Text += "\n";
}
}
```

აქ, ფუნქციის გამოცხადებისას უნდა მიეთითოს ან მხოლოდ სვეტების რაოდენობა:

```
int Minimumi(int mas[][3]),
```

ან სტრიქონებისა და სვეტების რაოდენობა:

```
int Minimumi(int mas[3][3]).
```

ქვემოთ მოყვანილ პროგრამაში Minimumi() ფუნქციას გადაეცემა ერთგანზომილებიანი CLR მასივი. ფუნქცია პოულობს და აბრუნებს მასივის მინიმალური ელემენტის მნიშვნელობას.

```
//      ფუნქციისათვის ერთგანზომილებიანი CLR მასივის გადაცემის დემონსტრირება
 int Minimumi(array<System::Int32>^    mas) {
int min_ricxvi = mas[0];
for ( int indexi =1; indexi < mas->Length; indexi++ )
        if  (min_ricxvi > mas[indexi] ) min_ricxvi = mas[indexi];
return min_ricxvi;
}
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
label2->Text = "";
array<Int32>^    masivi = gcnew array<Int32>(5) { 5, -1, 8, -4, 2 };

int shedegi = Minimumi(masivi);                    //        Minimumi ფუნქციას masivi მასივი გადაეცემა
        label1->Text = shedegi.ToString();
for each ( int ricxvi in masivi )
        label2->Text += ricxvi.ToString() + "  ";
}
```

აქ, პროგრამის int shedegi = Minimumi(masivi); სტრიქონში ხდება Minimumi() ფუნქციის გამოძახება. მას გადაეცემა masivi მასივის საწყისი მისამართი (პირველი ელემენტის მისამართი), რომელიც მიენიჭება mas პარამეტრს. შედეგად, masivi და mas მასივები მეხსიერების ერთსა და იმავე უბანს მიმართავენ. ამიტომ, მასივთან მუშაობა შესრულდება იმ უბანში, რომელიც masivi მასივისთვის გამოიყო.

ქვემოთ მოცემულ პროგრამაში Jami() ფუნქციას გადაეცემა ორგანზომილებიანი CLR მასივი. ფუნქცია ანგარიშობს და აბრუნებს მასივის ელემენტების ჯამს. ამ შემთხვევაშიც, ფუნქციას გადაეცემა მისი პირველი ელემენტის მისამართი.

```
//        ფუნქციისათვის ორგანზომილებიანი CLR მასივის გადაცემის დემონსტრირება
int Jami( int str_raod, int sv_raod, array<int, 2>^ mas) {
int jami = 0;
for ( int striqoni = 0; striqoni < str_raod; striqoni++ )
        for ( int sveti = 0; sveti < sv_raod; sveti++ )
                jami += mas[striqoni, sveti];
return jami;
}
//
private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = "";
int striqonebis_raodenoba = Convert::ToInt32(textBox1->Text);
int svetebis_raodeboba = Convert::ToInt32(textBox2->Text);
array<int, 2>^ masivi = gcnew array<int, 2>(striqonebis_raodenoba, svetebis_raodeboba);
//        masivi მასივის შევსება რიცხვებით
for ( int striqoni = 0; striqoni < striqonebis_raodenoba; striqoni++ )
        for ( int sveti = 0; sveti < svetebis_raodeboba; sveti++ )
                masivi[striqoni, sveti] = striqoni + sveti;
int jami = Jami(striqonebis_raodenoba, svetebis_raodeboba, masivi);
//        masivi მასივის ეკრანზე გამოტანა
for ( int striqoni = 0; striqoni < striqonebis_raodenoba; striqoni++ )
{
        for ( int sveti = 0; sveti < svetebis_raodeboba; sveti++ )
                label1->Text += masivi[striqoni, sveti].ToString() + "  ";
        label1->Text += "\n";
}
label2->Text = jami.ToString();
}
```

პარამეტრების გადაცემა მიმთითებლებისა და მიმართვების გამოყენებით

ფუნქციისთვის პარამეტრების გადაცემისას შეგვიძლია მიმთითებლებისა და მიმართვების გამოყენება. შევადგინოთ ფუნქცია, რომელსაც ორი პარამეტრი გადაეცემა. პარამეტრები წარმოადგენენ მთელ რიცხვზე მიმთითებლებს. ფუნქცია ამ პარამეტრებს მნიშვნელობებს უცვლის.

```
void Gacvla(int* par1, int* par2) {
int temp;
temp = *par1;
*par1 = *par2;
*par2 = temp;
}
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
label1->Text = L"პირველი რიცხვი = " + ricxvi1.ToString() + L"    მეორე რიცხვი = " +
ricxvi2.ToString();
Gacvla(&ricxvi1, &ricxvi2);
label2->Text = L"პირველი რიცხვი = " + ricxvi1.ToString() + L"    მეორე რიცხვი = " +
ricxvi2.ToString();
}
```

Gacvla ფუნქციის პარამეტრები მთელ რიცხვზე მიმთითებლებია. ამიტომ, ძირითადი პროგრამიდან ამ ფუნქციის გამოძახებისას მის par1 და par2 პარამეტრებს, შესაბამისად გადაეცემა ricxvi1 და ricxvi2 ცვლადების მისამართები. შედეგად, ამ პარამეტრების მნიშვნელობების შეცვლა გამოიწვევს ricxvi1 და ricxvi2 ცვლადების მნიშვნელობების შეცვლას. ეს ხდება იმიტომ, რომ მონაცემების დამუშავება შესრულდა იმ უბანში, რომელიც ricxvi1 და ricxvi2 ცვლადებს გამოეყო.

როგორც ვიცით, return ოპერატორი გასცემს ერთ შედეგს და ამთავრებს ფუნქციის შესრულებას. იმისათვის, რომ ფუნქციიდან მივიღოთ ერთზე მეტი მნიშვნელობა უნდა გამოვიყენოთ მიმთითებლები. მოყვანილი ფუნქცია return ოპერატორის მეშვეობით გასცემს მართკუთხედის პერიმეტრის, ხოლო მიმთითებლის მეშვეობით კი - ფართობის მნიშვნელობას.

```
int FartPerim(int gv1, int gv2, int* pfartobi) {
*pfartobi = gv1 * gv2;
return 2 * (gv1 + gv2);
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int gverdi1 = Convert::ToInt32(textBox1->Text);
int gverdi2 = Convert::ToInt32(textBox2->Text);
int fartobi, perimetri;
//        ფუნქციის გამოძახება
perimetri = FartPerim(gverdi1, gverdi2, &fartobi);
label1->Text = L"ფართობი = " + fartobi.ToString() + L"    პერიმეტრი = " + perimetri.ToString();
}
```

ამ შემთხვევაში, pfartobi მიმთითებელს გადაეცემა fartobi ცვლადის მისამართი. ამიტომ, pfartobi მისამართით ჩაწერილი მნიშვნელობა მიენიჭება fartobi ცვლადს.

ახლა, შევადგინოთ ფუნქცია, რომელიც return ოპერატორის გამოყენებით მართკუთხედის პერიმეტრს დაგვიბრუნებს, ხოლო მიმართვის გამოყენებით კი - ფართობს.

```
int FartPerim1(int gv1, int gv2, int& fart) {
fart = gv1 * gv2;
return 2 * (gv1 + gv2);
}
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int gverdi1 = Convert::ToInt32(textBox1->Text);
int gverdi2 = Convert::ToInt32(textBox2->Text);
int fartobi, perimetri;
//        ფუნქციის გამოძახება
perimetri = FartPerim1(gverdi1, gverdi2, fartobi);
label1->Text = L"ფართობი = " + fartobi.ToString() + L"    პერიმეტრი = " + perimetri.ToString();
}
```

ფუნქციას გამოძახებისას სამი არგუმენტი გადაეცემა. მესამეა fartobi ცვლადი, რომლის ფსევდონიმი ხდება fart პარამეტრი. ე.ი., მეხსიერების უბანს, რომელიც fartobi ცვლადისთვის გამოიყო, დაერქვა მეორე სახელი - fart. ამიტომ, fart ცვლადის მნიშვნელობის შეცვლა შეცვლის fartobi ცვლადის მნიშვნელობას, რადგან ცვლადების დამუშავება ერთსა და იმავე უბანში ხდება.

მოყვანილი პროგრამით ხდება ორი რიცხვის მნიშვნელობების გაცვლა. ამ შემთხვევაში, ფუნქციის პარამეტრები მიმართვებს წარმოადგენენ.

```
void Gacvla1(int& par1, int& par2) {
int temp;
temp = par1;
par1 = par2;
par2 = temp;
}
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
label1->Text = L"პირველი რიცხვი = " + ricxvi1.ToString() + L"    მეორე რიცხვი = " +
ricxvi2.ToString();
//        ფუნქციის გამოძახება
Gacvla1(ricxvi1, ricxvi2);
label2->Text = L"პირველი რიცხვი = " + ricxvi1.ToString() + L"    მეორე რიცხვი = " +
ricxvi2.ToString();
}
```

ფუნქციის გამოძახებისას, ricxvi1 ცვლადის ფსევდონიმი ხდება par1, ხოლო ricxvi2 ცვლადის ფსევდონიმი კი - par2. პარამეტრების მნიშვნელობების შეცვლა გამოიწვევს ricxvi1 და ricxvi2 არგუმენტების მნიშვნელობების შეცვლას.

თავი 6. კლასები, ინკაფსულაცია, ფუნქციები

კლასის გამოცხადება. ინკაფსულაცია

კლასი არის ობიექტზე ორიენტირებული დაპროგრამების ძირითადი სტრუქტურა. იგი ობიექტის შექმნის მექანიზმია. მის საფუძველზე იქმნება ობიექტი - ავტომობილი, თვითმფრინავი, მაღაზია, გეომეტრიული ფიგურა, სტუდენტი, ლექტორი, უნივერსიტეტი, ავადმყოფი და ა.შ. ობიექტი კლასის ეგზემპლარია. განსხვავება კლასსა და ობიექტს შორის ისაა, რომ კლასი არის ლოგიკური აბსტრაქცია, ობიექტი კი - ამ კლასის კონკრეტული რეალიზება კომპიუტერის მეხსიერებაში.

კლასში შემავალ ფუნქციას, ცვლადს, მასივს, მიმთითებელსა და კლასის სხვა ობიექტს კლასის წევრი ეწოდება. კლასი შეიძლება შეიცავდეს, აგრეთვე, სტატიკურ წევრებს, კონსტრუქტორებსა და დესტრუქტორებს. ამ ეტაპზე განვიხილავთ კლასის ძირითად წევრებს - ცვლადებს და ფუნქციებს.

კლასის გამოცხადების სინტაქსია:

მიმართვის_მოდიფიკატორი **class** ***კლასის_სახელი***

{

// ***კლასის დახურული ცვლადები და ფუნქციები***

public :

// ***კლასის ღია ცვლადები და ფუნქციები***

} [*ობიექტების_სია***];**

ობიექტების_სია შეიცავს მოცემული კლასის ტიპის მქონე ობიექტების სახელებს.

კორექტულად დაწერილ პროგრამაში კლასი განსაზღვრავს მხოლოდ ერთ ლოგიკურ ერთეულს. მაგალითად, კლასი, რომელიც შეიცავს ინფორმაციას ავტომანქანების შესახებ, არ უნდა შეიცავდეს სხვა ინფორმაციას, მაგალითად, შენობების ან ცხოველების შესახებ.

კლასი ორ ძირითად ფუნქციას ასრულებს, რომლებიც მონაცემების ინკაფსულაციას უზრუნველყოფენ. პირველი, ის მონაცემებს აკავშირებს კოდთან, რომელიც მათზე მანიპულირებს და მეორე, კლასი უზრუნველყოფს კლასის წევრებთან მიმართვის საშუალებებს. ***ინკაფსულაცია*** არის კლასის წევრებთან მიმართვის უფლებამოსილების გამიჯვნა, ანუ კლასის ზოგიერთ წევრს შეგვიძლია მივმართოთ კლასის გარეთ განსაზღვრული კოდიდან, ზოგიერთს კი - არა.

კლასის შიგნით გამოცხადებულ ცვლადებს და ფუნქციებს ამ კლასის ***წევრები*** (members) ეწოდება. ავტომატურად (ნაგულისხმევად), კლასში გამოცხადებული ცვლადები და ფუნქციები დახურულია (პრივატულია, private), ამიტომ მათი გამოცხადებისთვის არ არის აუცილებელი private სიტყვის გამოყენება. ღია (public) ცვლადებისა და ფუნქციების გამოცხადებისათვის გამოიყენება public სიტყვა.

კლასის ღია წევრებთან მიმართვა შეიძლება შესრულდეს ამ კლასის გარეთ განსაზღვრული კოდიდან. კლასის დახურულ წევრებთან მიმართვა შეუძლია მხოლოდ ამავე კლასის ფუნქციებს. ამ კლასის გარეთ განსაზღვრულ კოდს მხოლოდ ამავე კლასის ღია ფუნქციების გამოყენებით შეუძლია მიმართოს ამ კლასის დახურულ წევრებს.

კლასის წევრებთან მიმართვის შეზღუდვა არის ობიექტზე ორიენტირებული დაპროგრამების ერთ-ერთი ძირითადი პრინციპი. ის ობიექტებს იცავს არასწორი გამოყენებისაგან. კლასის დახურულ წევრებთან მიმართვის უფლებას ვაძლევთ მხოლოდ კლასის შიგნით განსაზღვრულ ფუნქციებს, რითაც თავიდან ვიცილებთ მონაცემებისთვის არაკორექტული მნიშვნელობების მინიჭებას კლასის გარეთ განსაზღვრული კოდით.

შევქმნათ Samkutxedi კლასი და მასში მოვათავსოთ სამი ღია ცვლადი - სამკუთხედის

გვერდები და ორი დახურული ცვლადი - პერიმეტრი და ფართობი:

```
class Samkutxedi {
int perimetri;                  //      დახურული ცვლადები
double fartobi;
public :
int gverdi1;                    //      ღია ცვლადები
int gverdi2;
int gverdi3;
};
```

როგორც მაგალითიდან ჩანს, ჩვენ შეგვიძლია პირდაპირ მივმართოთ სამკუთხედის გვერდებს, რადგან ისინი ღია წევრებია. რადგან perimetri და fartobi დახურული ცვლადებია, ამიტომ მათთვის მნიშვნელობების მინიჭება შეუძლია ამავე კლასში განსაზღვრულ ფუნქციებს. თუ perimetri-ს გამოვაცხადებთ ღია ცვლადად, მაშინ შეიძლება დავუშვათ შეცდომა. კერძოდ, თუ გვერდებს მივანიჭეთ მნიშვნელობები 1, 5 და 8, მაშინ perimetri ცვლადის მნიშვნელობა უნდა იყოს 14. რადგან perimetri ღია ცვლადია, ამიტომ ჩვენ შეგვიძლია მას ნებისმიერი მნიშვნელობა მივანიჭოთ, მაგალითად, 20. ამ შემთხვევაში მივიღებთ არასწორ სამკუთხედს. ამრიგად, პროგრამაში perimetri და fartobi ცვლადები იმიტომ არის პრივატული, რომ დაცული იყოს ობიექტის მთლიანობა. თუ მათ გამოვაცხადებთ ღია ცვლადად, მაშინ იარსებებს იმის საფრთხე, რომ ამ ცვლადებს არასწორი მნიშვნელობა მიენიჭოთ და შედეგად, სამკუთხედის გვერდების ჯამი შეიძლება არ დაემთხვეს პერიმეტრის მნიშვნელობას. იგივე ეხება სამკუთხედის ფართობს.

class სიტყვის გამოყენებით იქმნება მონაცემების ახალი ტიპი, რომლის სახელია Samkutxedi. მას გამოვიყენებთ Samkutxedi ტიპის ობიექტის შესაქმნელად. უნდა გვახსოვდეს, რომ class სიტყვის გამოყენებით კლასის გამოცხადებისას ხდება მხოლოდ მისი აღწერა, მაგრამ ფიზიკურად ობიექტი ამ დროს არ იქმნება. Samkutxedi ტიპის ობიექტი იქმნება შემდეგნაირად:

```
Samkutxedi Sam1;
```

ამ დროს მეხსიერებაში იქმნება Samkutxedi კლასის ეგზემპლარი (instance) - Sam1 ობიექტი. Sam1 ობიექტს ექნება Samkutxedi კლასის ყველა წევრის ასლი.

ამრიგად, ობიექტი შეიცავს კლასის თითოეული წევრის ასლს. ობიექტის წევრებთან მიმართვისათვის ვიყენებთ წერტილი (.) ოპერატორს. მისი სინტაქსია:

ობიექტის_სახელი.წევრის_სახელი

იმისათვის, რომ Sam1 ობიექტის gverdi2 ცვლადს მივანიჭოთ მნიშვნელობა 20, მინიჭების ოპერატორი შემდეგნაირად უნდა დავწეროთ:

```
Sam1.gverdi2 = 20;
```

კლასებთან მუშობისას უნდა გვახსოვდეს, რომ კლასის აღწერა ყოველთვის ცალკე ფაილში უნდა მოვათავსოთ. ამისათვის, ჯერ უნდა შევქმნათ ახალი პროექტი. შემდეგ, შევასრულოთ Project მენიუს Add New Item ბრძანება. გახსნილი ფანჯრის Templates: ზონაში მოვნიშნოთ Header File (.h) ელემენტი, Name: ველში შევიტანოთ ფაილის სახელი, მაგალითად, Klasebi და დავაჭიროთ Add კლავიშს. გახსნილ ფანჯარაში შეგვაქვს კლასის აღწერა:

```
//      კლასის აღწერა სხვა ფაილში უნდა იყოს
//      Klasebi.h ფაილი
class Samkutxedi {
        int perimetri;                  //      დახურული ცვლადები
        double fartobi;
public :
        int gverdi1;                    //      ღია ცვლადები
        int gverdi2;
```

```
        int gverdi3;
};
```

ამ ფაილში შეგვიძლია მოვათავსოთ იმდენი კლასი, რამდენიც გვჭირდება. შემდეგ, გავხსნათ Form1.h ჩანართი (button კლავიშზე ორჯერ დაწკაპუნებით) და ამ ფაილის დასაწყისში #pragma once სტრიქონის შემდეგ შევიტანოთ #include "Klasebi.h" დირექტივა. #include დირექტივა მითითებულ ფაილს ათავსებს მოცემულ ფაილში და შედეგად, მითითებულ ფაილში გამოცხადებული კლასები, ფუნქციები, ჩამოთვლები, დელეგატები და ცვლადები ხილული ხდება მიმდინარე ფაილის შიგნით. ამის შემდეგ, შეგვაქვს ძირითადი პროგრამა ისე, როგორც ამას ადრე ვაკეთებდით:

```
//      ობიექტის ცვლადებთან მუშაობის დემონსტრირება
//      Form1.h ფაილი
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e){
Samkutxedi Sam1;
int perimetri;

//      Sam1 ობიექტის ცვლადებს ენიჭებათ მნიშვნელობები
Sam1.gverdi1 = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2 = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3 = Convert::ToInt32(textBox3->Text);
//      სამკუთხედის პერიმეტრის გამოთვლა
perimetri = Sam1.gverdi1 + Sam1.gverdi2 + Sam1.gverdi3;
label1->Text = L"სამკუთხედის პერიმეტრია  " + perimetri.ToString();
}
```

პროგრამაში გამოცხადებული perimetri ცვლადი და კლასში გამოცხადებული პრივატული perimetri ცვლადი სხვადასხვა ცვლადებია და მეხსიერების სხვადასხვა უბანშია მოთავსებული. ჯერ-ჯერობით, კლასის პრივატულ ცვლადთან მიმართვას ვერ შევძლებთ. ამას შევძლებთ ფუნქციების შესწავლის შემდეგ.

შემდგომში, მთელ წიგნში, კლასის აღწერა და პროგრამის კოდი ერთად იქნება მოთავსებული, მაგრამ თქვენ კომპიუტერში პროგრამის შეტანისას კლასის აღწერას ცალკე ფაილში მოათავსებთ, პროგრამას კი - Form1.h ფაილში.

როგორც აღვნიშნეთ, ობიექტების შექმნის ძირითადი პრინციპია ის, რომ თითოეულ ობიექტს აქვს კლასის ცვლადების საკუთარი ასლი. შედეგად, ერთი ობიექტის ცვლადები დამოუკიდებელია მეორე ობიექტის ცვლადებისაგან, ამიტომ მათი მნიშვნელობები შეიძლება ერთმანეთისაგან განსხვავდებოდეს. მაგალითად, თუ არსებობს Samkutxedi კლასის ორი ობიექტი, მაშინ თითოეულ მათგანს ექნება gverdi1, gverdi2, gverdi3, perimetri და fartobi ცვლადების საკუთარი ასლები, რომლებიც მეხსიერების სხვადასხვა უბანში იქნება მოთავსებული და შესაბამისად, მათი მნიშვნელობები შეიძლება ერთმანეთისაგან განსხვავდებოდეს. ქვემოთ მოცემული პროგრამით ხდება ამ პრინციპის დემონსტრირება.

```
//      ორი ობიექტის ცვლადებთან მუშაობის დემონსტრირება
//      Klasebi.h ფაილი
class Samkutxedi {
int perimetri;                          //      დახურული ცვლადები
double fartobi;
public : int gverdi1;                   //      ღია ცვლადები
int gverdi2;
int gverdi3;
};
```

```
//        Form1.h ფაილი
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Samkutxedi Sam1;                          //        იქმნება Sam1 ობიექტი
Samkutxedi Sam2;                          //        იქმნება Sam2 ობიექტი
int Sam1_perimetri, Sam2_perimetri;

//        Sam1 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam1.gverdi1 = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2 = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3 = Convert::ToInt32(textBox3->Text);

//        Sam2 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam2.gverdi1 = Convert::ToInt32(textBox4->Text);
Sam2.gverdi2 = Convert::ToInt32(textBox5->Text);
Sam2.gverdi3 = Convert::ToInt32(textBox6->Text);

//        თითოეული სამკუთხედის პერიმეტრის გამოთვლა
Sam1_perimetri = Sam1.gverdi1 + Sam1.gverdi2 + Sam1.gverdi3;
Sam2_perimetri = Sam2.gverdi1 + Sam2.gverdi2 + Sam2.gverdi3;
label1->Text = Sam1_perimetri.ToString();
label2->Text = Sam2_perimetri.ToString();
}
```

ობიექტები შეგვიძლია კლასის აღწერისას გამოვაცხადოთ. მოცემულ პროგრამაში კლასის აღწერის შემდეგ გამოცხადებულია Sam1 და Sam2 ობიექტები.

```
//        Klasebi.h ფაილი
class Samkutxedi {
int perimetri;                            //        დახურული ცვლადები
double fartobi;
public : int gverdi1;                     //        ღია ცვლადები
int gverdi2;
int gverdi3;
} Sam1, Sam2;
//        Form1.h ფაილი
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int Sam1_perimetri, Sam2_perimetri;

//        Sam1 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam1.gverdi1 = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2 = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3 = Convert::ToInt32(textBox3->Text);

//        Sam2 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam2.gverdi1 = Convert::ToInt32(textBox4->Text);
Sam2.gverdi2 = Convert::ToInt32(textBox5->Text);
Sam2.gverdi3 = Convert::ToInt32(textBox6->Text);

//        თითოეული სამკუთხედის პერიმეტრის გამოთვლა
```

```
Sam1_perimetri = Sam1.gverdi1 + Sam1.gverdi2 + Sam1.gverdi3;
Sam2_perimetri = Sam2.gverdi1 + Sam2.gverdi2 + Sam2.gverdi3;
label1->Text = Sam1_perimetri.ToString();
label2->Text = Sam2_perimetri.ToString();
}
```

ობიექტების მასივი

მასივი შეიძლება შეიცავდეს ობიექტებსაც. მოცემული პროგრამით ხდება ობიექტების მასივთან მუშაობის დემონსტრირება.

```
//      ობიექტების მასივთან მუშაობის დემონსტრირება
ref class Samkutxedi {
int perimetri;                          //      დახურული ცვლადები
double fartobi;
public : int gverdi1;                   //      ღია ცვლადები
int gverdi2;
int gverdi3;
};
//
private: System::Void button7_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text  = "";
//      იქმნება 5 ობიექტისაგან შემდგარი masivi მასივი
array<Samkutxedi^>^ masivi = gcnew array<Samkutxedi^>(5);
int indexi;
//      იქმნება 5 სამკუთხედი
for ( indexi = 0; indexi < 5; indexi++ ) {
        masivi[indexi] = gcnew Samkutxedi;
}
//      თითოეული სამკუთხედის გვერდებისთვის მნიშვნელობების მინიჭება
for ( indexi = 0; indexi < 5; indexi++ ) {
        masivi[indexi]->gverdi1 = indexi + 10;
        masivi[indexi]->gverdi2 = indexi + 20;
        masivi[indexi]->gverdi3 = indexi + 40;
}
//      თითოეული სამკუთხედის გვერდების მნიშვნელობების ეკრანზე გამოტანა
for ( indexi = 0; indexi < 5; indexi++ )
label1->Text += (indexi+1).ToString() + L"  სამკუთხედის გვერდებია: " +
                masivi[indexi]->gverdi1.ToString() + "   " + masivi[indexi]->gverdi2.ToString() +
                "   " + masivi[indexi]->gverdi3.ToString() + "\n";
//      თითოეული სამკუთხედის გვერდების მნიშვნელობების ეკრანზე გამოტანა
indexi = 1;
for each ( Samkutxedi^ obieqti in masivi )
label2->Text += (indexi++).ToString() + L"  სამკუთხედის გვერდებია: " +
                obieqti->gverdi1.ToString() + "   " + obieqti->gverdi2.ToString() +
                "   " + obieqti->gverdi3.ToString() + "\n";
}
```

ობიექტებზე მიმთითებელი

აქამდე, ობიექტის წევრებს მივმართავდით წერტილი (.) ოპერატორის საშუალებით. ობიექტის წევრებს შეგვიძლია მივმართოთ მიმთითებლის საშუალებითაც. ამ შემთხვევაში, გამოიყენება ისარი (->) ოპერატორი. ობიექტზე მიმთითებელი განისაზღვრება ისევე, როგორც ნებისმიერი ტიპის ცვლადზე მიმთითებელი. ამისათვის კლასის სახელის შემდეგ ვუთითებთ ობიექტის სახელს, რომლის წინ ვარსკვლავია (*) მოთავსებული. მაგალითად,

```
Samkutxedi *obj1;
```

ობიექტის მისამართის მისაღებად, მისი სახელის წინ უნდა დავწეროთ & ოპერატორი:

```
Samkutxedi obj1;
Samkutxedi *misamarti;
misamarti = &obj1;
```

თუ ობიექტზე მიმთითებლის მნიშვნელობას 1-ით გავზრდით, მაშინ ის მომდევნო ობიექტს მიმართავს.

მოცემული პროგრამით ხდება ობიექტზე მიმთითებელთან მუშაობის დემონსტრირება.

```
//      ობიექტზე მიმთითებელთან მუშაობის დემონსტრირება
class ChemiKlasi {
public :
        int ricxvi1;
        int ricxvi2;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ChemiKlasi obj1;                //      ობიექტის შექმნა
ChemiKlasi *mimtitebeli;        //      მიმთითებლის გამოცხადება

mimtitebeli = &obj1;            //      მიმთითებლისთვის obj1 ობიექტის მისამართის მინიჭება
mimtitebeli->ricxvi1 = Convert::ToInt32(textBox1->Text);
obj1.ricxvi2 = Convert::ToInt32(textBox2->Text);
label1->Text = mimtitebeli->ricxvi1.ToString();
label2->Text = obj1.ricxvi2.ToString();
}
```

პროგრამაში ChemiKlasi *mimtitebeli; გამოცხადება ქმნის ChemiKlasi ტიპის ობიექტზე მიმთითებელს. ამ დროს ობიექტი არ იქმნება. mimtitebeli მიმთითებელს obj1 ობიექტის მისამართს შემდეგნაირად ვანიჭებთ:

```
mimtitebeli = &obj1;
```

მიმთითებლის საშუალებით obj1 ობიექტის წევრთან მიმართვისათვის ისარი (->) ოპერატორი გამოიყენება:

```
mimtitebeli->ricxvi1 = Convert::ToInt32(textBox1->Text);
```

ობიექტების მინიჭება

თუ ორ ობიექტს ერთნაირი ტიპი აქვს, მაშინ ერთი ობიექტი შეიძლება მეორეს მივანიჭოთ. ამ დროს, ობიექტს ენიჭება მეორე ობიექტის ყველა წევრის მნიშვნელობების ასლი.

ეს ეხება მასივსაც. მიუხედავად იმისა, რომ ორივე ობიექტი ერთნაირი იქნება, ისინი მაინც სხვადასხვა ობიექტებად რჩებიან და მეხსიერების სხვადასხვა უბანს იკავებენ. მაგალითი:

```
//      ერთი ტიპის მქონე ობიექტების მინიჭება
ref class ChemiKlasi {
public :
        int mteli;
        double wiladi;
        bool logikuri;
static array<int>^ masivi = gcnew array<int> (5);
};
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = ""; label2->Text = "";
int indexi;
ChemiKlasi^ obj1 = gcnew ChemiKlasi();
ChemiKlasi^ obj2;
//      obj1 ობიექტის წევრებს ენიჭება მნიშვნელობები
obj1->mteli =  Convert::ToInt32(textBox1->Text);
obj1->wiladi = Convert::ToDouble(textBox2->Text);
obj1->logikuri = Convert::ToBoolean(textBox3->Text);
for ( indexi = 0; indexi < obj1->masivi->Length; indexi++ )
        obj1->masivi[indexi] = indexi + 10;
//      obj2 ობიექტს ენიჭება obj1 ობიექტი
obj2 = obj1;
//      obj2 ობიექტის წევრების ეკრანზე გამოტანა
for ( indexi = 0; indexi < obj2->masivi->Length; indexi++ )
        label2->Text += obj2->masivi[indexi].ToString() + "   ";
label1->Text = obj2->mteli.ToString() + "   " + obj2->wiladi.ToString() + "   " + obj2->logikuri.ToString();
}
```

ობიექტების მინიჭებისას, ობიექტების ტიპები არა მარტო ფიზიკურად უნდა იყოს ერთნაირი, არამედ ტიპების სახელებითაც. მაგალითი:

```
//      ერთი ტიპის ობიექტს ენიჭება მეორე ტიპის ობიექტი
class ChemiKlasi1 {
public :
        int mteli;
        double wiladi;
};
class ChemiKlasi2 {
public :
        int mteli;
        double wiladi;
};
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
ChemiKlasi1 obj1;
ChemiKlasi1 obj3;
ChemiKlasi2 obj2;

obj1.mteli = Convert::ToInt32(textBox1->Text);
```

```
obj1.wiladi = Convert::ToDouble(textBox2->Text);
//      შეცდომა! სხვადასხვა ტიპის მქონე ობიექტების მინიჭება
obj2 = obj1;
obj3 = obj1;
label1->Text = obj1.mteli.ToString() + "   " + obj1.wiladi.ToString();
label2->Text = obj3.mteli.ToString() + "   " + obj3.wiladi.ToString();
}
```

აქ obj2 = obj1; მინიჭება დაუშვებელია, რადგან ამ ობიექტებს სხვადასხვა ტიპი აქვს, თუმცა ეს ტიპები ფიზიკურად ერთნაირია. obj3 = obj1; მინიჭება დასაშვებია, რადგან ამ ობიექტების ტიპები ერთნაირია.

ფუნქცია

ფუნქცია შეიძლება იყოს ან არ იყოს კლასის წევრი. ფუნქცია, რომელიც არ არის კლასის წევრი, არის გლობალური. კლასის შიგნით გამოცხადებულ ფუნქციას ***მეთოდი*** ეწოდება. მას ***ფუნქცია-წევრსაც*** (*member function*) უწოდებენ. კლასის შიგნით უნდა გამოვაცხადოთ ფუნქციის პროტოტიპი. ***ფუნქციის პროტოტიპი*** არის ფუნქციის სათაური, რომელიც შედგება ფუნქციის მიერ გაცემული ტიპის, ფუნქციის სახელისა და პარამეტრების სიისაგან. ფუნქციის პროტოტიპის გამოცხადების მაგალითი:

```
int Funqcia(int par1, double par2, System::String^ par3);
```

Funqcia() ფუნქცია გასცემს int ტიპის შედეგს და მთელი, წილადი და სტრიქონული ტიპის სამი პარამეტრი აქვს.

ფუნქციის ***კოდი (ტანი)*** შეიძლება განსაზღვრული იყოს როგორც კლასის შიგნით, ისე კლასის გარეთ. კლასის შიგნით განსაზღვრული ფუნქციის სინტაქსია:

მიმართვის_მოდიფიკატორი : შედეგის_ტიპი ფუნქციის_სახელი(პარამეტრების_სია)

```
{
ფუნქციის კოდი
}
```

აქ ***მიმართვის_მოდიფიკატორი*** იღებს public მნიშვნელობას. თუ ის არ არის მითითებული, მაშინ ფუნქცია დახურულია და შესაბამისად, მისაწვდომი იქნება მხოლოდ იმ კლასის შიგნით, რომელშიც ის არის განსაზღვრული.

ზემოთ მოცემულ მაგალითებში სამკუთხედის პერიმეტრის გამოთვლა სრულდებოდა ძირითად პროგრამაში. პერიმეტრი უმჯობესია გამოითვალოს უშუალოდ კლასის შიგნით, რადგან მისი მნიშვნელობა დამოკიდებულია სამკუთხედის გვერდების ზომებზე. სამივე ეს სიდიდე კი ინკაფსულირებულია Samkutxedi კლასში. გარდა ამისა, Samkutxedi კლასისთვის იმ ფუნქციის დამატება, რომელშიც სრულდება პერიმეტრის გამოთვლა აუმჯობესებს ამ კლასის ობიექტზე ორიენტირებულ სტრუქტურას. მაგალითი:

```
//      სამკუთხედის პერიმეტრის გამოთვლა კლასის შიგნით გამოცხადებულ ფუნქციაში
class Samkutxedi {
        int perimetri;
        double fartobi;
public :
        int gverdi1;
        int gverdi2;
        int gverdi3;
//      perimetri ფუნქცია განსაზღვრულია Samkutxedi კლასში
```

```
void Perimetri() {
perimetri = gverdi1 + gverdi2 + gverdi3;
System::Windows::Forms::MessageBox::Show(L"სამკუთხედის პერიმეტრია - " + perimetri.ToString());
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Samkutxedi Sam1;
Samkutxedi Sam2;
//        Sam1 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam1.gverdi1  = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2  = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3  = Convert::ToInt32(textBox3->Text);
//        პერიმეტრის გამოთვლა პირველი სამკუთხედისათვის
Sam1.perimetri ();
//        Sam2 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam2.gverdi1  = Convert::ToInt32(textBox4->Text);
Sam2.gverdi2  = Convert::ToInt32(textBox5->Text);
Sam2.gverdi3  = Convert::ToInt32(textBox6->Text);
//        პერიმეტრის გამოთვლა მეორე სამკუთხედისათვის
Sam2.Perimetri();
}
```

Perimetri() ფუნქცია განსაზღვრულია როგორც public, რაც იძლევა მისი გამოძახების შესაძლებლობას პროგრამის ნებისმიერი ადგილიდან. void სიტყვა მიუთითებს იმას, რომ Perimetri() ფუნქცია არ აბრუნებს (არ გასცემს) მნიშვნელობას. შემდეგ, ფიგურულ ფრჩხილებში მოთავსებულია ფუნქციის კოდი, რომელშიც გამოითვლება სამკუთხედის პერიმეტრი. მისი მნიშვნელობა ეკრანზე გამოაქვს MessageBox::Show() ფუნქციას. რადგან, Samkutxedi ტიპის თითოეულ ობიექტს აქვს gverdi1, gverdi2 და gverdi3 ცვლადების საკუთარი ასლები, ამიტომ Perimetri() ფუნქციის გამოძახებისას პერიმეტრის გამოსათვლელად გამოყენებული იქნება გამომძახებელი ობიექტის ცვლადები.

Sam1.Perimetri(); სტრიქონში გამომძახებელია Sam1 ობიექტი. აქ ხდება Sam1 ობიექტის Perimetri() ფუნქციის გამოძახება, რის შემდეგ მართვა ამ ფუნქციას გადაეცემა. ის Sam1 ობიექტის ცვლადებთან იმუშავებს. როდესაც ფუნქცია მუშაობას დაამთავრებს, მართვა გადაეცემა პროგრამის იმ ადგილს, საიდანაც მისი გამოძახება მოხდა. პროგრამის შესრულება გაგრძელდება კოდის იმ სტრიქონიდან, რომელიც მოსდევს ფუნქციის გამომძახებელ სტრიქონს. ჩვენ შემთხვევაში, Sam1.Perimetri() ფუნქციის გამოძახება გამოიწვევს პერიმეტრის მნიშვნელობის გამოტანას Sam1 სამკუთხედისათვის, Sam2.Perimetri() ფუნქციის გამოძახება კი - პერიმეტრის მნიშვნელობის გამოტანას Sam2 სამკუთხედისათვის.

თუ ფუნქციის კოდი გვინდა განვსაზღვროთ კლასის გარეთ, მაშინ კლასის სახელის შემდეგ უნდა მოვათავსოთ "::" სიმბოლოები და ფუნქციის სახელი. "**::**" სიმბოლოებს ***ხილვადობის უბნის გაფართოების ოპერატორი*** ეწოდება. ფუნქციის განსაზღვრა შეგვიძლია მოვათავსოთ კლასის განსაზღვრის შემდეგ იმავე ფაილში:

```
// კლასის გარეთ ფუნქციის გამოცხადების დემონსტრირება
class Samkutxedi {
int perimetri;
double fartobi;
public :
```

```
int gverdi1;
int gverdi2;
int gverdi3;
//        Perimetri ფუნქციის პროტოტიპის გამოცხადება
void Perimetri();
};
//        Perimetri() ფუნქცია განსაზღვრულია Samkutxedi კლასის გარეთ
void Samkutxedi::Perimetri() {
perimetri = gverdi1 + gverdi2 + gverdi3;
System::Windows::Forms::MessageBox::Show(L"სამკუთხედის პერიმეტრია - " + perimetri.ToString());
}
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e){
Samkutxedi Sam1;
Samkutxedi Sam2;
//        Sam1 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam1.gverdi1  = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2  = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3  = Convert::ToInt32(textBox3->Text);
//        პერიმეტრის გამოთვლა პირველი სამკუთხედისათვის
Sam1.Perimetri();
//        Sam2 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam2.gverdi1  = Convert::ToInt32(textBox4->Text);
Sam2.gverdi2  = Convert::ToInt32(textBox5->Text);
Sam2.gverdi3  = Convert::ToInt32(textBox6->Text);
//        პერიმეტრის გამოთვლა მეორე სამკუთხედისათვის
Sam2.Perimetri();
}
```

კლასის შიგნით public void Perimetri(); სტრიქონში ხდება Perimetri() ფუნქციის პროტოტიპის გამოცხადება. რადგან, Perimetri() ფუნქცია განსაზღვრულია კლასის გარეთ, ამიტომ მისი სახელის წინ ვწერთ კლასის სახელს და „::“ ოპერატორს.

პრივატული ცვლადის გამოტანა შეგვიძლია, აგრეთვე, Perimetri() ფუნქციისთვის პარამეტრად label1 კომპონენტის გადაცემის გზით. მაგალითი:

```
//        ფუნქციისთვის label კომპონენტის გადაცემა
class Samkutxedi {
private : int perimetri;
public : int gverdi1;
        int gverdi2;
        int gverdi3;
//        perimetri ფუნქციის განსაზღვრა
void Perimetri(System::Windows::Forms::Label^ lab1) {
perimetri = gverdi1 + gverdi2 + gverdi3;
lab1->Text = L"სამკუთხედის პერიმეტრია - " + perimetri.ToString();
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
Samkutxedi Sam1;

//        Sam1 ობიექტის ცვლადებს ენიჭება მნიშვნელობები
Sam1.gverdi1  = Convert::ToInt32(textBox1->Text);
Sam1.gverdi2  = Convert::ToInt32(textBox2->Text);
Sam1.gverdi3  = Convert::ToInt32(textBox3->Text);
//        პერიმეტრის გამოთვლა პირველი სამკუთხედისათვის
Sam1.Perimetri(label1);
}
```

როგორც ვხედავთ, Perimetri() ფუნქციის გამოძახებისას, მას პარამეტრად label1 კომპონენტი გადაეცემა. მისი მისამართი მიენიჭება lab1 პარამეტრს, რომლის ტიპია Label. ამის შემდეგ, lab1 პარამეტრი მუშაობს ზუსტად ისე, როგორც label1 კომპონენტი. შედეგად, პერიმეტრის მნიშვნელობა გამოჩნდება label1 კომპონენტში.

ფუნქციიდან მართვის დაბრუნება

ახლა Perimetri()ფუნქცია გადავაკეთოთ ისე, რომ მან გამოთვალოს პერიმეტრი და დაუბრუნოს ის გამომძახებელ პროგრამას. ასეთი მიდგომის ერთ-ერთი უპირატესობაა ის, რომ დაბრუნებული მნიშვნელობა შეგვიძლია გამოვიყენოთ გამოთვლებში. ქვემოთ მოცემულ პროგრამაში Perimetri() ფუნქციას ეკრანზე აღარ გამოაქვს პერიმეტრის მნიშვნელობა, არამედ ახდენს მის გამოთვლას და შედეგის დაბრუნებას.

```
//        return ოპერატორის გამოყენების დემონსტრირება
class Samkutxedi {
        int perimetri;
public : int gverdi1;
        int gverdi2;
        int gverdi3;
//        ფუნქციის განსაზღვრა
int Perimetri() {
        perimetri = gverdi1 + gverdi2 + gverdi3;
        return perimetri;                //        მნიშვნელობის დაბრუნება
}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
//        tolgverda და tolferda ობიექტების შექმნა
Samkutxedi tolgverda;
Samkutxedi tolferda;
int tolgverda_perimetri, tolferda_perimetri;

//        tolgverda ობიექტის ცვლადებს მნიშვნელობები ენიჭება
tolgverda.gverdi1 = Convert::ToInt32(textBox1->Text);
tolgverda.gverdi2 = Convert::ToInt32(textBox2->Text);
tolgverda.gverdi3 = Convert::ToInt32(textBox3->Text);

//        tolferda ობიექტის ცვლადებს მნიშვნელობები ენიჭება
```

```
tolferda.gverdi1 = Convert::ToInt32(textBox4->Text);
tolferda.gverdi2 = Convert::ToInt32(textBox5->Text);
tolferda.gverdi3 = Convert::ToInt32(textBox6->Text);

//      tolgverda და tolferda სამკუთხედებისათვის პერიმეტრი გამოითვლება
tolgverda_perimetri = tolgverda. Perimetri ();
tolferda_perimetri  = tolferda. Perimetri ();
label1->Text = tolgverda_perimetri.ToString();
label2->Text = tolferda_perimetri.ToString();
}
```

პროგრამაში Perimetri() ფუნქციის მიერ გაცემული მნიშვნელობები შესაბამისად მიენიჭება tolgverda_perimetri და tolferda_perimetri ცვლადებს.

პარამეტრის გამოყენება

როგორც ვნახეთ, გამოძახებისას ფუნქციას შეგვიძლია გადავცეთ ერთი ან მეტი პარამეტრი. ქვემოთ მოცემულ პროგრამაში ფუნქციისათვის მნიშვნელობის გადასაცემად პარამეტრი გამოიყენება. ArisLuwi კლასის შიგნით განსაზღვრული luwi_kenti(int par1) ფუნქცია აბრუნებს bool ტიპის შედეგს. თუ მისთვის გადაცემული მნიშვნელობა არის ლუწი, მაშინ ის გასცემს true მნიშვნელობას, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას.

```
//      პროგრამა განსაზღვრავს რიცხვი კენტია თუ ლუწი
class ArisLuwi {
// ფუნქციის განსაზღვრა, რომელიც ამოწმებს რიცხვი კენტია თუ ლუწი
public : bool luwi_kenti( int par1) {
if ( (par1 % 2 ) == 0 ) return true;
        else return false;
}
};
//
private: System::Void button6_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi;
ArisLuwi obieqti;
label1->Text = "";

ricxvi = Convert::ToInt32(textBox1->Text);
if ( obieqti.luwi_kenti(ricxvi) )
label1->Text = L" რიცხვი " + ricxvi.ToString() + L" ლუწია";
        else  label1->Text = L" რიცხვი " + ricxvi.ToString() + L" კენტია";
}
```

luwi_kenti() ფუნქციის გამოძახებისას ricxvi არგუმენტის მნიშვნელობა მიენიჭება par1 პარამეტრს.

კიდევ ერთი მაგალითი. Gamyofi კლასი შეიცავს arisGamyofi() ფუნქციას, რომელიც განსაზღვრავს არის თუ არა პირველი პარამეტრი მეორე პარამეტრის გამყოფი.

```
//      პროგრამა განსაზღვრავს ერთი რიცხვი არის თუ არა მეორის გამყოფი
class Gamyofi {
```

```
//        ფუნქცია ამოწმებს  პირველი პარამეტრი უნაშთოდ იყოფა თუ არა მეორე პარამეტრზე
public : bool arisGamyofi(int par1, int par2) {
if ( ( par1 % par2 ) == 0 ) return true;
        else return false;
}
};
//
private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e) {
label2->Text = "";
Gamyofi obieqti;

int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
if ( obieqti.arisGamyofi(ricxvi1, ricxvi2) )
        label2->Text = ricxvi1.ToString() + L" არის " + ricxvi2.ToString() + L"-ის გამყოფი";
else    label2->Text = ricxvi1.ToString() + L" არ არის " + ricxvi2.ToString() + L"-ის გამყოფი";
}
```

arisGamyofi() ფუნქციის გამოძახებისას ricxvi1 არგუმენტის მნიშვნელობა მიენიჭება par1პარამეტრს, ricxvi2 არგუმენტის მნიშვნელობა კი - par2 პარამეტრს.

ქვემოთ მოცემულ პროგრამაში Minimumi() ფუნქციას გადაეცემა CLR-მასივი. ფუნქცია პოულობს და აბრუნებს მასივის მინიმალური ელემენტის მნიშვნელობას.

```
//        ფუნქციისათვის CLR-მასივის გადაცემის დემონსტრირება
class MinSidide {
//        ფუნქცია გასცემს მასივის მინიმალური ელემენტის მნიშვნელობას
public : int Minimumi(array<System::Int32>^    mas) {
int min_ricxvi = mas[0];
for ( int indexi =1; indexi < mas->Length; indexi++ )
if  (min_ricxvi > mas[indexi] ) min_ricxvi = mas[indexi];
return min_ricxvi;
}
};
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
label2->Text = "";
array<Int32>^    masivi = gcnew array<Int32>(5) { 5, -1, 8, -4, 2 };
MinSidide obieqti;

int shedegi = obieqti.Minimumi(masivi);        //        Minimumi ფუნქციას masivi მასივი გადაეცემა
        label1->Text = shedegi.ToString();
for each ( int ricxvi in masivi )
        label2->Text += ricxvi.ToString() + "  ";
}
```

პროგრამის int shedegi = obieqti.Minimumi(masivi); სტრიქონში ხდება Minimumi() ფუნქციის გამოძახება. მას გადაეცემა masivi მასივის საწყისი მისამართი (პირველი ელემენტის მისამართი), რომელიც მიენიჭება mas პარამეტრს. შედეგად, masivi და mas მასივები მეხსიერების ერთსა და იმავე უბანს მიმართავენ. ამრიგად, მთელი მასივის გადაცემის ნაცვლად გადაიცემა მისი მისამართი.

კონსტრუქტორი

კონსტრუქტორი (constructor) არის ფუნქცია, რომლის დანიშნულებაა ობიექტის ცვლადების ინიციალიზება. მისი გამოცხადების სინტაქსი ჩვეულებრივი ფუნქციის სინტაქსის მსგავსია. კონსტრუქტორი ყოველთვის გამოიძახება ობიექტის შექმნისას. ჩვეულებრივი ფუნქციისაგან განსხვავებით:

- კონსტრუქტორს ისეთივე სახელი აქვს, როგორიც კლასს.
- დასაბრუნებელი მნიშვნელობის ტიპი კონსტრუქტორში არ ეთითება.

კონსტრუქტორის სინტაქსია:

კლასის_სახელი()
{
კონსტრუქტორის კოდი
}

როგორც წესი, კონსტრუქტორები გამოიყენება ობიექტის ცვლადებისათვის საწყისი მნიშვნელობების მისანიჭებლად, ან ინიციალიზების ნებისმიერი სხვა პროცედურის შესასრულებლად, რომლებიც აუცილებელია ობიექტის შესაქმნელად.

ყველა კლასს აქვს კონსტრუქტორი მიუხედავად იმისა, ის განსაზღვრულია თუ არა. C++ ენაში გათვალისწინებულია კონსტრუქტორის არსებობა, რომელიც ობიექტის ყველა ცვლადს ანიჭებს ნულოვან (ეს ეხება ჩვეულებრივი ტიპის ცვლადებს) ან null მნიშვნელობას (ეს ეხება მიმართვითი ტიპის ცვლადებს). ასეთ კონსტრუქტორს *ავტომატური კონსტრუქტორი* (გაჩუმებითი კონსტრუქტორი) ეწოდება. ის იმ შემთხვევაში გამოიძახება, როდესაც კლასში კონსტრუქტორი განსაზღვრული არ არის. თუ კონსტრუქტორი აშკარადაა განსაზღვრული კლასში, მაშინ სწორედ ის იქნება გამოძახებული.

გლობალური ობიექტებისათვის კონსტრუქტორი გამოიძახება მაშინ, როდესაც პროგრამა მუშობას იწყებს, ლოკალური ობიექტებისათვის კი - ცვლადის გამოცხადებისას.

პროფესიულ დონეზე შედგენილ პროგრამაში ობიექტის ცვლადების ინიციალიზება ყოველთვის კონსტრუქტორის მიერ სრულდება. მაგალითად, აქამდე განხილულ პროგრამებში ობიექტის ცვლადებს მნიშვნელობებს ჩვენ ვანიჭებდით. უმჯობესია, ეს კონსტრუქტორმა გააკეთოს. მაგალითი:

```
//      კონსტრუქტორთან მუშაობის დემონსტრირება
class Samkutxedi {
int perimetri;
public : int gverdi1;
         int gverdi2;
         int gverdi3;
//       კონსტრუქტორი პარამეტრებით
Samkutxedi(int par1, int par2, int par3) {
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3;
}
//       ფუნქცია გასცემს პერიმეტრს
int Perimetris_Gacema() {
return perimetri;
```

```
}
};
//
private: System::Void button7_Click(System::Object^  sender, System::EventArgs^  e) {
int g1 = Convert::ToInt32(textBox1->Text);
int g2 = Convert::ToInt32(textBox2->Text);
int g3 = Convert::ToInt32(textBox3->Text);
int g4 = Convert::ToInt32(textBox4->Text);
int g5 = Convert::ToInt32(textBox5->Text);
int g6 = Convert::ToInt32(textBox6->Text);
Samkutxedi Sam1(g1, g2, g3);
Samkutxedi Sam2(g4, g5, g6);

int SamkutxedisPerimetri1 = Sam1.Perimetris_Gacema();
int SamkutxedisPerimetri2 = Sam2.Perimetris_Gacema();
label1->Text = L"პირველი სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri1.ToString();
label2->Text = L" მეორე სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri2.ToString();
}
```

პროგრამაში Samkutxedi Sam1(g1, g2, g3); სტრიქონში სრულდება Samkutxedi() კონსტრუქტორის გამოძახება. მას სამი პარამეტრი აქვს, რომლებიც გამოიყენება სამკუთხედის გვერდებისათვის მნიშვნელობების მისანიჭებლად. g1, g2 და g3 არგუმენტების მნიშვნელობები, შესაბამისად par1, par2 და par3 პარამეტრებს მიენიჭება. ისინი თავის მხრივ gverdi1, gverdi2 და gverdi3 ცვლადებს ენიჭება.

მოცემული პროგრამით ხდება ობიექტების გამოცხადება კლასის აღწერისას.

```
//      ობიექტების გამოცხადება კლასის განსაზღვრისას
class Samkutxedi {
        int gverdi1;
        int gverdi2;
        int gverdi3;
        int perimetri;
public :
Samkutxedi(int par1, int par2, int par3) {
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
}
int Perimetris_Gacema() {
perimetri = gverdi1 + gverdi2 + gverdi3;
return perimetri;
}
} Sam1(1, 2, 3), Sam2(11, 12, 13);
//
private: System::Void button10_Click(System::Object^  sender, System::EventArgs^  e) {
int g1 = Convert::ToInt32(textBox1->Text);
int g2 = Convert::ToInt32(textBox2->Text);
int g3 = Convert::ToInt32(textBox3->Text);
int g4 = Convert::ToInt32(textBox4->Text);
```

```
int g5 = Convert::ToInt32(textBox5->Text);
int g6 = Convert::ToInt32(textBox6->Text);
Samkutxedi Sam3(g1, g2, g3);
Samkutxedi Sam4(g4, g5, g6);

int SamkutxedisPerimetri1 = Sam1.Perimetris_Gacema();
int SamkutxedisPerimetri2 = Sam2.Perimetris_Gacema();
int SamkutxedisPerimetri3 = Sam3.Perimetris_Gacema();
int SamkutxedisPerimetri4 = Sam4.Perimetris_Gacema();

label1->Text = L" პირველი სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri1.ToString();
label2->Text = L" მეორე     სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri2.ToString();
label3->Text = L" მესამე     სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri3.ToString();
label4->Text = L" მეოთხე   სამკუთხედის პერიმეტრი = " + SamkutxedisPerimetri4.ToString();
}
```

კონსტრუქტორმა კლასში გამოცხადებული ცვლადების ინიციალიზებისთვის შეიძლება ***ინიციალიზების სია*** გამოიყენოს. ეს სია ეთითება ":" სიმბოლოს შემდეგ, რომელიც კონსტრუქტორის პარამეტრების გამოცხადებას მოსდევს. მოცემულ პროგრამაში gverdi1 ცვლადს par1 პარამეტრის მნიშვნელობა ენიჭება, gverdi2 ცვლადს - par2 პარამეტრის მნიშვნელობა, gverdi3 ცვლადს კი - par13 პარამეტრის მნიშვნელობა.

```
//        ინიციალიზების სიის გამოყენების დემონსტრირება
class Samkutxedi {
        int gverdi1;
        int gverdi2;
        int gverdi3;
        int perimetri;
public :
Samkutxedi(int par1, int par2, int par3) : gverdi1(par1), gverdi2(par2), gverdi3(par3) {
perimetri = gverdi1 + gverdi2 + gverdi3;
}
int Perimetris_Gacema() {
return perimetri;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi obj(cvladi1, cvladi2, cvladi3);
label1->Text = obj.Perimetris_Gacema().ToString();
}
```

როდესაც კონსტრუქტორს ერთი არგუმენტი აქვს, მაშინ შეგვიძლია გამოვიყენოთ ინიციალიზების ორი ფორმა:

```
//        პირველი ფორმა
{
```

```
...
ChemiKlasi obj(5);
...
}
და
//        მეორე ფორმა
{
...
ChemiKlasi obj = 5;
...
}
```

ინიციალიზების მეორე ფორმის აზრი ის არის, რომ ერთარგუმენტიანი კონსტრუქტორისთვის ის იძლევა ამ არგუმენტის ტიპის არაცხადი გარდაქმნის ორგანიზების საშუალებას იმ კლასის ტიპად, რომელსაც კონსტრუქტორი ეკუთვნის. მაგალითი:

```
//        კონსტრუქტორის მიერ ცვლადების ინიციალიზების
//        ორივე ფორმის დემონსტრირება
class ChemiKlasi {
        int cvladi;
public :
        ChemiKlasi(int par) { cvladi = par; }
        int Naxva() { return cvladi; }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Char simbolo = Convert::ToChar(textBox3->Text);
ChemiKlasi obj1(ricxvi1);                //        ინიციალიზების პირველი ფორმა
ChemiKlasi obj2 = ricxvi2;               //        ინიციალიზების მეორე ფორმა

label1->Text = obj1.Naxva().ToString();
label2->Text = obj2.Naxva().ToString();
//        აქ სრულდება ტიპის ავტომატური გარდაქმნა
obj1 = simbolo;
label3->Text = obj1.Naxva().ToString();
}
```

არაცხადი გარდაქმნა შეიძლება ავკრძალოთ **explicit** (ცხადი) სპეციფიკატორის მეშვეობით. ის გამოიყენება მხოლოდ კონსტრუქტორების მიმართ. ავტომატური გარდაქმნა ასეთი ტიპის კონსტრუქტორებისათვის არ სრულდება. მაგალითი:

```
//        არაცხადი გარდაქმნის აკრძალვის დემონსტრირება
class ChemiKlasi1 {
        int cvladi;
public :
        explicit ChemiKlasi1(int par) { cvladi = par; }
        int Naxva() { return cvladi; }
};
//
```

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);
Char simbolo = Convert::ToChar(textBox2->Text);
ChemiKlasi1 obj1 = ricxvi;

label1->Text = obj1.Naxva().ToString();
obj1 = simbolo;
label2->Text = obj1.Naxva().ToString();
}
```

class და struct ტიპები

C++/CLI ენას აქვს თავისი საკუთარი struct და class ტიპები. ისინი საშუალებას გვაძლევენ განვსაზღვროთ შემდეგი სამომხმარებლო სტრუქტურები და კლასები:

- **value struct** (მნიშვნელობითი ტიპის სტრუქტურა, მნიშვნელობის ტიპის სტრუქტურა),
- **value class** (მნიშვნელობითი ტიპის კლასი, მნიშვნელობის ტიპის კლასი),
- **ref struct** (მიმართვითი სტრუქტურა),
- **ref class** (მიმართვითი კლასი).

მშობლიური C++ ენის მსგავსად C++/CLI ენაში ერთადერთი განსხვავება სტრუქტურებსა და კლასებს შორის იმაში მდგომარეობს, რომ სტრუქტურების წევრები ავტომატურად არიან ღია (public), კლასების წევრები კი - პრივატული (private). არსებითი განსხვავება მნიშვნელობის (მნიშვნელობითი) ტიპის კლასებსა და მიმართვითი ტიპის კლასებს შორის იმაში მდგომარეობს, რომ მნიშვნელობის ტიპის ცვლადები შეიცავენ მონაცემებს, მიმართვითი ტიპის ცვლადები კი - მისამართებს, რადგან არიან დესკრიპტორები.

მშობლიური C++ ენისგან განსხვავებით C++/CLI ენაში:

ა) კლასის ფუნქცია-წევრი არ შეიძლება გამოვაცხადოთ როგორც const;

ბ) this მიმთითებელი T ტიპის მნიშვნელობითი კლასების არასტატიკურ ფუნქცია-წევრებში არის interior_ptr<T> ტიპის შიგა მიმთითებელი, ხოლო T ტიპის მიმართვით კლასებში კი - T^ ტიპის დესკრიპტორი. ეს გარემოება უნდა გავითვალისწინოთ C++/CLI ფუნქციიდან this მიმთითებლის დაბრუნების ან ლოკალურ ცვლადში მისი შენახვისას.

მნიშვნელობით კლასებსა და მიმართვით კლასებს ედებათ შემდეგი შეზღუდვები:

- მნიშვნელობითი კლასი ან მიმართვითი კლასი არ შეიძლება შეიცავდეს ველებს, რომლებიც წარმოადგენენ მშობლიური C++ ენის მასივებს ან მშობლიური C++ ენის კლასების ტიპებს.
- მნიშვნელობით კლასებსა და მიმართვით კლასებში დაუშვებელია მეგობრული ფუნქციები.
- მნიშვნელობით კლასებსა და მიმართვით კლასებში დაუშვებელია ბიტობრივი ველები.

CLI პროგრამებში საბაზო ტიპების სახელები (მაგალითად, int, double და ა.შ.) წარმოადგენენ შემოკლებებს მნიშვნელობითი კლასების ტიპებისათვის. როდესაც ვაცხადებთ მნიშვნელობითი კლასის ტიპის ცვლადს, მაშინ მისთვის მეხსიერება სტეკში გამოიყოფა. თუ გვინდა, რომ მნიშვნელობითი კლასის ტიპის ცვლადი (ობიექტი) გროვაში შევქმნათ, მაშინ უნდა გამოვიყენოთ gcnew ოპერატორი. ამ შემთხვევაში, ცვლადი, რომელიც მნიშვნელობითი კლასის ტიპის ობიექტს მიმართავს, უნდა იყოს დესკრიპტორი. მაგალითი:

```
int mteli1 = 55;                 //     mteli1 ცვლადია და 55 ინახება სტეკში
int^ mteli2 = 24;                //     mteli2 დესკრიპტორია და 24 ინახება გროვაში
int^ mteli3 = gcnew int(50);     //     mteli3 დესკრიპტორია და 50 ინახება გროვაში
```

ეს ცვლადები შეგვიძლია გამოვიყენოთ არითმეტიკის ოპერაციებში, ოღონდ mteli2 და mteli3 დესკრიპტორების სახელების წინ უნდა მივუთითოთ * ოპერაცია:

```
{
int   mteli1 = Convert::ToInt32(textBox1->Text);
int^ mteli3 = Convert::ToInt32(textBox2->Text);
int^ mteli2 = gcnew int(Convert::ToInt32(textBox3->Text));
int shedegi = mteli1 + *mteli2 + *mteli3;
label1->Text = shedegi.ToString();
}
```

მნიშვნელობითი კლასის ტიპი

მნიშვნელობითი კლასი არის კლასის შედარებით მარტივი ტიპი. ის შეგვიძლია გამოვიყენოთ ახალი მარტივი ტიპის შესაქმნელად, რომელსაც გამოვიყენებთ საბაზო ტიპების ანალოგიურად. მნიშვნელობითი კლასის ტიპის ცვლადი სტეკში იქმნება და თავის მნიშვნელობას უშუალოდ ინახავს. თუმცა, ჩვენ შეგვიძლია გამოვიყენოთ თვალყურის სადევნებელი დესკრიპტორი იმისათვის, რომ მივმართოთ მნიშვნელობითი კლასის ტიპის მონაცემს, რომელიც CLR გროვაში ინახება. მოცემული პროგრამით ხდება მარტივი მნიშვნელობითი კლასის განსაზღვრის დემონსტრირება.

```
//        მნიშვნელობით კლასთან მუშაობის დემონსტრირება
value class Samkutxedi {
private :
int gverdi1;
int gverdi2;
int gverdi3;
int perimetri;
double fartobi;
public :
Samkutxedi(int par1, int par2, int par3) {
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3 ;
fartobi = ( gverdi1 * gverdi2 ) / 2;
}
double Naxva_Fartobi() {
return fartobi;
}
int NaxvaPerimetri() {
return perimetri;
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
```

```
int cvladi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi obj1(cvladi1, cvladi2, cvladi3);
Samkutxedi obj2 = Samkutxedi(cvladi1, cvladi2, cvladi3);
Samkutxedi^ obj3 = Samkutxedi(cvladi1, cvladi2, cvladi3);
Samkutxedi obj4 = *obj3;
Samkutxedi obj5;

label1->Text =  L"სამკუთხედის პერიმეტრი = " + obj1.NaxvaPerimetri().ToString() +
                L"\nსამკუთხედის ფართობი = " + obj1.Naxva_Fartobi().ToString();
label2->Text =  L"სამკუთხედის ფართობი = " + obj2.Naxva_Fartobi().ToString();
label3->Text =   L"სამკუთხედის პერიმეტრი = " + obj3->NaxvaPerimetri().ToString();
label4->Text =  L"სამკუთხედის პერიმეტრი = " + obj4.NaxvaPerimetri().ToString();
}
```

პროგრამაში obj1, obj2, obj4 და obj5 ობიექტების ტიპია Samkutxedi და ისინი სტეკში ინახება. რაც შეეხება obj3 ობიექტს, ის არის Samkutxedi^ ტიპის დესკრიპტორი და შესაბამისად, CLR გროვაში ინახება. obj4 არის იმ ობიექტის ასლი, რომელსაც obj3 დესკრიპტორი მიმართავს.

```
Samkutxedi obj4 = *obj3;
```

მინიჭების შემდეგ obj4 შეიცავს იმ ობიექტის დუბლიკატს, რომელსაც obj3 დესკრიპტორი მიმართავს. მნიშვნელობითი კლასის ტიპის ცვლადები შეიცავენ უნიკალურ ობიექტებს, ამიტომ ორი ასეთი ცვლადი არ შეიძლება ერთსა და იმავე ობიექტს მიმართავდეს. მნიშვნელობითი კლასის ტიპის ერთი ცვლადის მინიჭება ასეთივე ტიპის მეორე ცვლადისთვის ყოველთვის გულისხმობს მნიშვნელობების გადაწერას. რამდენიმე დესკრიპტორს შეუძლია მიმართოს ერთ ობიექტს მეხსიერებაში და შეიძლება ერთი დესკრიპტორის მნიშვნელობის მინიჭება მეორე დესკრიპტორისთვის მისამართის ან nullptr-ის გადაწერის გზით.

პროგრამაში obj5 ობიექტის ცვლადები ინიციალიზებული იქნება ნულებით. საქმე ის არის, რომ Samkutxedi კლასს არ აქვს უპარამეტრებო კონსტრუქტორი. უპარამეტრებო კონსტრუქტორის განსაზღვრა არ შეიძლება მნიშვნელობით კლასში. ამიტომ, მნიშვნელობით კლასში ავტომატურად არის ჩართული არაცხადი უპარამეტრებო კონსტრუქტორი, რომელიც ცვლადების ინიციალიზებას მოახდენს ნულით, ხოლო დესკრიპტორების ინიციალიზებას კი - nullptr მნიშვნელობით. obj5 ობიექტის ინიციალიზებისთვის გამოძახებული იქნება სწორედ არაცხადი უპარამეტრებო კონსტრუქტორი.

არსებობს კიდევ ორი შეზღუდვა მნიშვნელობითი კლასებისთვის:

- მნიშვნელობითი კლასის განსაზღვრაში არ უნდა ჩავრთოთ ასლის კონსტრუქტორი (მას მომდევნო თავში ვისწავლით);
- მნიშვნელობით კლასში არ შეიძლება მინიჭების ოპერატორის ხელახალი განსაზღვრა.

მნიშვნელობითი კლასების ობიექტები ყოველთვის გადაიწერება მათი ველების გადაწერის გზით. ასევე სრულდება ასეთი კლასის ერთი ობიექტისთვის მეორე ობიექტის მინიჭება. მნიშვნელობების კლასები, ჩვეულებრივ გამოიყენება მარტივი ობიექტების შესაქმნელად, რომლებიც მონაცემების მცირე რაოდენობას შეიცავენ. ისეთი ობიექტების შესაქმნელად, რომლებიც დიდი რაოდენობის მონაცემებს შეიცავენ და რომლებისთვისაც მნიშვნელობით კლასებზე არსებული შეზღუდვები პრობლემებს ქმნიან, გამოიყენება მიმართვითი კლასები (ref class).

მიმართვითი კლასის ტიპი

მიმართვით კლასს არ აქვს მნიშვნელობითი კლასისათვის დამახასიათებელი შეზღუდვები. მიმართვით კლასს მშობლიური C++ენის კლასისგან განსხვავებით არ აქვს ნაგულისხმევი ასლების კონსტრუქტორი ან ნაგულისხმევი მინიჭების ოპერაციები. თუ

მიმართვითი კლასი უნდა უზრუნველყოფდეს რომელიმე ამ ოპერაციას, მაშინ ჩვენ კლასში აშკარად უნდა განვსაზღვროთ შესაბამისი ფუნქცია. მიმართვითი კლასი განისაზღვრება ref class საკვანძო სიტყვის გამოყენებით. მოცემული პროგრამით ხდება მიმართვით კლასთან მუშაობის დემონსტრირება.

```
//          მიმართვით კლასთან მუშაობის დემონსტრირება
ref class Samkutxedi {
private :
        int gverdi1;
        int gverdi2;
        int gverdi3;
        int perimetri;
        double fartobi;
public :
//          უარგუმენტებო კონსტრუქტორი გვერდებს ანიჭებს გაჩუმებით მნიშვნელობებს
Samkutxedi() : gverdi1(10), gverdi2(20), gverdi3(30) {
fartobi = ( gverdi1 * gverdi2 ) / 2;
}
Samkutxedi(int par1, int par2, int par3) {
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3 ;
}
double Naxva_Fartobi() {
return fartobi;
}
int NaxvaPerimetri() {
return perimetri;
}
};
private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi^ obj1;
Samkutxedi^ obj2 = gcnew Samkutxedi(cvladi1, cvladi2, cvladi3);

obj1 = gcnew Samkutxedi;
label1->Text = L"სამკუთხედის ფართობი = " + obj1->Naxva_Fartobi().ToString();
label2->Text = L"სამკუთხედის პერიმეტრი = " + obj2->NaxvaPerimetri().ToString();
}
```

აქ obj1 არის Samkutxedi^ ტიპის დესკრიპტორი. ამ დროს ობიექტი არ იქმნება. obj1 დესკრიპტორს გაჩუმებით ენიჭება nullptr მნიშვნელობა. ამიტომ, ის არაფერზე არ მიუთითებს. მიმართვითი კლასის ტიპის ცვლადისაგან განსხვავებით მნიშვნელობის კლასის ტიპის ცვლადი ობიექტს ყოველთვის შეიცავს. obj2 არის Samkutxedi^ ტიპის ობიექტის დესკრიპტორი.

```
obj1 = gcnew Samkutxedi;
```

მინიჭების შესრულებისას გამოიძახება უპარამეტრებო კონსტრუქტორი. იქმნება Samkutxedi

ტიპის ობიექტი, რომლის მისამართი obj1 ცვლადში ინახება.

დესტრუქტორი და "ნაგვის შეგროვება"

როდესაც C++ ენაში არ ხდება ობიექტთან მიმართვა, მაშინ ის განიხილება როგორც შემდგომში არაგამოყენებადი და სისტემა ავტომატურად, პროგრამისტის მონაწილეობის გარეშე შლის ობიექტს, ანუ ათავისუფლებს ობიექტის მიერ დაკავებულ მეხსიერებას. ამ ოპერაციას "ნაგვის შეგროვება" ეწოდება. გამოთავისუფლებული მეხსიერება შემდგომში შეიძლება გამოყენებული იყოს სხვა ობიექტებისათვის.

"ნაგვის შეგროვება" პერიოდულად სრულდება პროგრამის მუშაობის განმავლობაში. ამ ოპერაციის დასაწყებად ორი პირობა უნდა შესრულდეს. ჯერ ერთი, უნდა არსებობდეს ობიექტები, რომლებიც შეიძლება წავშალოთ მეხსიერებიდან, და მეორე, ასეთი ოპერაციის აუცილებლობა. რადგან "ნაგვის შეგროვება" პროცესორის დროს იკავებს, ამიტომ ის მხოლოდ აუცილებლობის შემთხვევაში შესრულდება, ამასთან, პროგრამისტი ზუსტად ვერ განსაზღვრავს ამ მომენტს.

დესტრუქტორი (destructor) არის ფუნქცია, რომელიც გამოიძახება მეხსიერებიდან ობიექტის წასაშლისას. მისი სინტაქსია:

```
~კლასის_სახელი()
{
დესტრუქტორის კოდი
}
```

როგორც ვხედავთ, დესტრუქტორის სახელის წინ მითითებულია სიმბოლო ტილდა (~). დესტრუქტორის განსაზღვრისას დასაბრუნებელი მნიშვნელობის ტიპი არ ეთითება. ლოკალური ობიექტები იშლება მაშინ, როდესაც ისინი ხილვადობის უბნიდან გადიან, გლობალური ობიექტები კი - პროგრამის დამთავრებისას. კონსტრუქტორისა და დესტრუქტორის მისამართის მიღება შეუძლებელია.

დესტრუქტორის გამოძახება ხდება უშუალოდ "ნაგვის შეგროვების" ოპერაციის წინ მაშინ, როდესაც სრულდება მისი კლასის ობიექტის წაშლა მეხსიერებიდან. რადგან, წინასწარ არ ვიცით, თუ როდის დაიწყება "ნაგვის შეგროვების" პროცესი, ამიტომ არ გვეცოდინება თუ როდის შესრულდება დესტრუქტორის გამოძახება. თუ პროგრამამ მუშაობა დაამთავრა "ნაგვის შეგროვების" ოპერაციის დაწყებამდე, მაშინ დესტრუქტორი არასოდეს არ გამოიძახება.

მოცემული პროგრამით ხდება დესტრუქტორთან მუშაობის დემონსტრირება.

```
//        დესტრუქტორთან მუშაობის დემონსტრირება
ref class Klasi1 {
public :
int ricxvi1;
Klasi1(int par1) {
ricxvi1 = par1;
}
~Klasi1() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს დესტრუქტორი\nობიექტი წაიშალა");
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
```

```
Klasi1^ obj1 = gcnew Klasi1(cvladi);
label1->Text = obj1->ricxvi1.ToString();
delete obj1;
}
```

პროგრამის დამთავრების შემდეგ გაიცემა შეტყობინება - „ობიექტი წაიშალა“.

დესტრუქტორი და ფინალიზატორი მიმართვით კლასში

მიმართვითი კლასის დესტრუქტორის გამოცხადება ისევე ხდება, როგორ მშობლიურ C++ ენაში. მიმართვითი კლასის დესტრუქტორი გამოიძახება მაშინ, როდესაც დესკრიპტორი გადის ხილვადობის უბნიდან ან ობიექტი არის იმ ობიექტის ნაწილი, რომელიც იშლება. მიმართვითი კლასის დესკრიპტორის მიმართ შეგვიძლია გამოვიყენოთ delete ოპერაცია. შედეგად, შესრულდება დესტრუქტორის გამოძახება.

მიმართვით კლასებში დესტრუქტორები შეიძლება დაგვჭირდეს მაშინ, როდესაც კლასის ობიექტები იყენებენ სხვა რესურსებს, რომლებიც არ იმყოფებიან ნაგვის შემგროვებლის მართვის ქვეშ. ასეთი რესურსებია, მაგალითად ფაილები, რომლებიც უნდა დაიხურის ჩვეულებრივი გზით ობიექტის წაშლისას. ასეთი რესურსების გასუფთავება შეგვიძლია აგრეთვე, ***ფინალიზატორების*** (finalizer) გამოყენებით.

ფინალიზატორი არის მიმართვითი კლასის ფუნქცია-წევრი, რომელიც გამოიძახება ცხადად ან ობიექტის მიმართ delete ოპერაციის გამოყენების შედეგად. მემკვიდრე კლასებში ფინალიზატორები იმ მიმდევრობით გამოიძახება, რა მიმდევრობითაც უნდა მოხდეს დესტრუქტორების გამოძახება. ანუ ჯერ გამოიძახება საბაზო კლასის ფინალიზატორი, შემდეგ მემკვიდრე კლასების ფინალიზატორები იერარქიის მიხედვით და ბოლოს უკანასკნელი მემკვიდრე კლასის ფინალიზატორი.

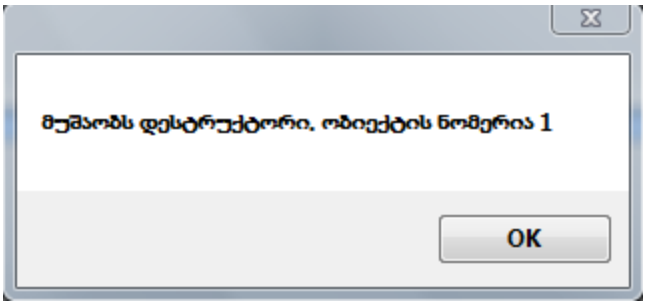
ფინალიზატორის გამოცხადება ხდება დესტრუქტორის მსგავსად, ოღონდ კლასის სახელის წინ ~ სიმბოლოს ნაცვლად უნდა მივუთითოთ ! სამბოლო.

მოცემული პროგრამით ხდება ფინალიზატორთან მუშაობის დემონსტრირება.

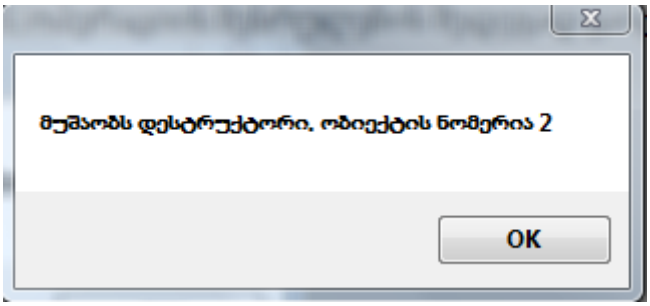
```
//      ფინალიზატორთან მუშაობის დემონსტრირება
public ref class Klasi {
public :
//      კონსტრუქტორი
Klasi(int par) : nomeri(par) { }
//      დესტრუქტორი
~Klasi() {
System::Windows::Forms::MessageBox::Show(
                        L"მუშაობს დესტრუქტორი. ობიექტის ნომერია " + nomeri.ToString());
}
//      ფინალიზატორი
!Klasi() {
System::Windows::Forms::MessageBox::Show(
                        L"მუშაობს ფინალიზატორი. ობიექტის ნომერია " + nomeri.ToString());
}
private :
        int nomeri;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Klasi^ obj1 = gcnew Klasi(ricxvi1);
Klasi^ obj2 = gcnew Klasi(ricxvi2);
Klasi^ obj3 = gcnew Klasi(ricxvi3);
delete obj1;
obj2->~Klasi();
}
```

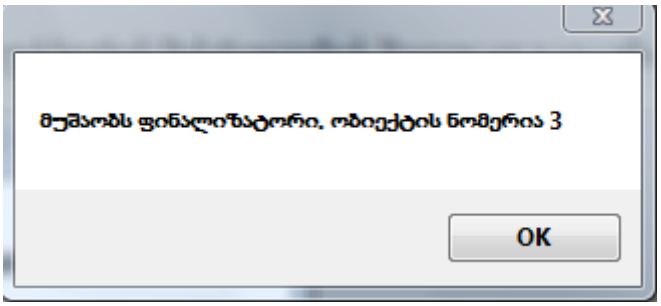
პროგრამაში delete obj1; ოპერაციის შესრულების შედეგად გაიცემა შეტყობინება:



obj2->~Klasi(); დესტრუქტორის შესრულების შედეგად გაიცემა შეტყობინება:



პროგრამის დამთავრების შემდეგ გაიცემა შეტყობინება:



როგორც ვხედავთ, დესტრუქტორები გამოიძახება მაშინ, როდესაც ობიექტი იშლება ან როდესაც დესტრუქტორი აშკარად გამოიძახება. ორივე ამ შემთხვევაში არ ხდება ობიექტების ფინალიზატორების გამოძახება. რადგან, obj3 ობიექტი იშლება ნაგვის შემგროვებლის მიერ პროგრამის დამთავრებისას, ამიტომ ფინალიზატორი გამოიძახება არამართვადი რესურსების გასუფთავების მიზნით.

ამრიგად, თუ კლასში განსაზღვრულია დესტრუქტორი და ფინალიზატორი, მაშინ მხოლოდ ერთ-ერთი მათგანი იქნება გამოძახებული ობიექტის წაშლისას. დესტრუქტორი

გამოიძახება იმ შემთხვევაში, როდესაც ობიექტს პროგრამულად ვშლით, ფინალიზატორი კი იმ შემთხვევაში, როდესაც ობიექტი თვითონ ქრება ან გადის ხილვადობის არედან. ამიტომ, თუ გვინდა, რომ გამოძახებული იყოს ფინალიზატორი არამართვადი რესურსების გასუფთავების მიზნით ობიექტის წაშლისას, მაშინ ეს ობიექტი აშკარად არ უნდა წავშალოთ.

თუ პროგრამაში გავაკომენტარებთ პროგრამის ორ უკანასკნელ სტრიქონს:

```
//delete obj1;
//obj2->~Klasi();
```

მაშინ პროგრამის დამთავრებისას გამოძახებული იქნება obj1 და obj2 ობიექტების ფინალიზატორებიც. თუ Klasi კლასში გავაკომენტარებთ ფინალიზატორს, მაშინ obj3 ობიექტის დესტრუქტორი არ იქნება გამოძახებული ნაგვის შემგროვებლის მიერ და არანაირი გასუფთავება არ შესრულდება.

ამრიგად, იმისათვის, რომ ობიექტის მიერ გამოყენებული არამართვადი რესურსები კორექტულად იყოს გათავისუფლებული, იმისგან დამოუკიდებლად თუ როგორ იშლება ობიექტი, კლასში უნდა განვსაზღვროთ დესტრუქტორიც და ფინალიზატორიც.

this მიმთითებელი

ფუნქციას გამოძახებისას ავტომატურად გადაეცემა არაცხადი არგუმენტი, რომელიც წარმოადგენს გამომძახებელ ობიექტზე მიმთითებელს. მას this მიმთითებელი ეწოდება. this მიმთითებელი გადაეცემა მხოლოდ ფუნქცია-წევრებს. მეგობრულ ფუნქციებს ის არ გადაეცემა. იმის გასაგებად, თუ როგორ მუშაობს this მიმთითებელი, განვიხილოთ პროგრამა, რომელშიც იქმნება Axarisxeba კლასი. ის განკუთვნილია რიცხვის ხარისხის გამოსათვლელად.

```
//        ხარისხის გამოთვლა სრულდება კონსტრუქტორის კოდში
ref class Axarisxeba {
public : double d;
          int i;
          double shedegi;
Axarisxeba(double par_ricxvi, int par_xarisxi) {
d = par_ricxvi;
i = par_xarisxi;
shedegi = 1;

if ( par_xarisxi == 0 ) return;
for ( ; par_xarisxi > 0; par_xarisxi-- ) shedegi *= d;
}
//        shedegi  შედეგის გაცემა
double xarisxis_dabruneba() {
return shedegi;
}
};
//
private: System::Void button8_Click(System::Object^  sender, System::EventArgs^  e) {
double ricxvi  = Convert::ToDouble(textBox1->Text);
int xarisxi = Convert::ToInt32(textBox2->Text);
Axarisxeba^ obieqti = gcnew Axarisxeba(ricxvi, xarisxi);
```

```
label1->Text = obieqti->d.ToString() + L" ხარისხად " + obieqti->i.ToString() +
                L" არის " + obieqti->xarisxis_dabruneba().ToString();
}
```

როგორც ვიცით ფუნქციის ფარგლებში კლასის სხვა წევრებთან მიმართვა შეძლება განხორციელდეს ობიექტის ან კლასის სახელის მითითების გარეშე, ე.ი. უშუალოდ. ამიტომ, xarisxis_dabruneba() ფუნქციის შიგნით return shedegi; ოპერატორის შესრულების შედეგად გაიცემა მოცემული ობიექტის shedegi ცვლადის მნიშვნელობა. მაგრამ, ეს ოპერატორი შეგვიძლია ასეც ჩავწეროთ:

```
return this->shedegi;
```

აქ this მიმთითებელი მიუთითებს იმ ობიექტზე, რომელსაც xarisxis_dabruneba() ფუნქცია ეკუთვნის. this->shedegi ცვლადი არის მოცემული ობიექტის shedegi ცვლადი. მაგალითად, თუ xarisxis_dabruneba() ფუნქცია გამოძახებულია obj1 ობიექტისთვის, მაშინ this მიმთითებელი მიუთითებს obj1 ობიექტზე. ოპერატორის ჩაწერა this სიტყვის გარეშე არის ჩაწერის შემოკლებული ფორმა.

C++ ენის სინტაქსი იძლევა ერთნაირი სახელების გამოყენების საშუალებას პარამეტრებისა და ლოკალური ცვლადებისათვის. ასეთ შემთხვევაში, პარამეტრი მალავს ლოკალურ ცვლადს და მასთან მიმართვა შესაძლებელია მხოლოდ this მიმთითებლის გამოყენებით. ქვემოთ მოცემული პროგრამით ხდება დამალულ ცვლადთან მიმართვა this სიტყვის გამოყენებით.

```
//          დამალულ ცვლადთან მიმართვის დემონსტრირება
ref class Axarisxeba {
public : double ricxvi;
         int xarisxi;
double shedegi;
Axarisxeba(double ricxvi, int xarisxi) {
this->ricxvi = ricxvi;              // ობიექტის this.ricxvi ცვლადს ენიჭება ricxvi პარამეტრი
this->xarisxi = xarisxi;            // ობიექტის this.xarisxi ცვლადს ენიჭება xarisxi პარამეტრი

shedegi = 1;
if ( this->xarisxi == 0 ) return;
for ( ; this->xarisxi > 0; this->xarisxi-- ) shedegi *= this->ricxvi;
}
};
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
double ricxvi1 = Convert::ToDouble(textBox1->Text);
int   xarisxi1 = Convert::ToInt32(textBox2->Text);
Axarisxeba^ obieqti = gcnew Axarisxeba(ricxvi1, xarisxi1);
label1->Text = obieqti->shedegi.ToString();
}
```

Axarisxeba() კონსტრუქტორის ამ ვერსიაში პარამეტრების სახელები და ობიექტის ცვლადების სახელები ერთნაირია, ამიტომ ობიექტის ცვლადები დამალული იქნება. this მიმთითებლის გამოყენებით შესაძლებელი ხდება მათთან მიმართვა.

კლასის სტატიკური წევრი

კლასის შიგნით ცვლადები შეგვიძლია გამოვაცხადოთ როგორც სტატიკური (static). თუ კლასში გამოცხადებულია სტატიკური ცვლადი, მაშინ იარსებებს მისი მხოლოდ ერთი ასლი მიუხედავად იმისა, თუ ამ კლასის რამდენ ობიექტს შევქმნით. ეს ობიექტები ერთობლივად გამოიყენებენ ამ ერთ ცვლადს. როგორც ცნობილია, იქმნება ჩვეულებრივი ცვლადის იმდენი ასლი, რამდენი ობიექტიც არის შექმნილი და ამ ცვლადებთან მიმართვა შესაძლებელია შესაბამისი ობიექტების მეშვეობით. კლასის სტატიკურ წევრთან მიმართვა შესაძლებელია მისი სახელის წინ კლასის სახელისა და ხილვადობის უბნის გაფართოების ოპერატორის მითითების გზით.

სინამდვილეში, კლასის სტატიკური ცვლადი არის გლობალური, რომლის ხილვადობის უბანი შემოსაზღვრულია იმ კლასით, რომელშიც ის არის განსაზღვრული. C++ენაში სტატიკური ცვლადების შემოტანის ძირითადი მიზანი იყო გლობალური ცვლადების გამოყენებაზე უარის თქმა. საქმე ის არის, რომ გლობალური ცვლადების გამოყენება არღვევს ობიექტზე ორიენტირებული დაპროგრამების ინკაფსულირების პრინციპს.

```
//      ღია და დახურულ სტატიკურ ცვლადებთან მუშაობის დემონსტრირება
//      Klasi.h ფაილი
ref class Klasi {
        static int statikuri_cvladi1;
public :
        static int statikuri_cvladi2;
        void Minicheba(int par) {
                statikuri_cvladi1 = par;
        }
        int Naxva() { return statikuri_cvladi1; }
};
//      Form1.h ფაილი
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj1 = gcnew Klasi();
int ricxvi1 = Convert::ToInt32(textBox1->Text);
//      Klasi1 კლასის statikuri_cvladi1 დახურული
//      სტატიკური ცვლადისთვის მნიშვნელობის მინიჭება
obj1->Minicheba(ricxvi1);
label1->Text = obj1->Naxva().ToString();
//      მიმართვა Klasi1 კლასის statikuri_cvladi2 ღია სტატიკურ ცვლადთან
Klasi::statikuri_cvladi2 = Convert::ToInt32(textBox2->Text);
label2->Text = Klasi::statikuri_cvladi2.ToString();
}
```

კლასის სტატიკურ ცვლადს გამოცხადებისას ავტომატურად ენიჭება ნულოვანი მნიშვნელობა. კლასის სტატიკურ წევრებს არ აქვთ this მიმართვა.

კლასში სტატიკური შეიძლება იყოს ფუნქციაც. სტატიკური ფუნქცია არ შეიძლება იყოს გამოცხადებული როგორც const (მუდმივა) და volatile (ცვლადი). სტატიკური ფუნქცია შეიძლება გამოძახებული იყოს ამ კლასის ობიექტის მეშვეობით, აგრეთვე, კლასის სახელისა და ხილვადობის უბნის გაფართოების ოპერატორის მეშვეობით. მაგალითი:

```
//      სტატიკურ ფუნქციასთან მუშაობის დემონსტრირება
//      Klasi.h ფაილი
ref class Klasi1 {
        static int statikuri_cvladi;
```

```
        int chveulebrivi_cvladi;
public :
        static void statikuri_Minicheba1(int par) {
                statikuri_cvladi = par;
        }
        static void statikuri_Minicheba2(Klasi1^ obj, int par) {
                obj->chveulebrivi_cvladi = par + 10;
        }
        int Naxva1() { return statikuri_cvladi; }
        int Naxva2() { return chveulebrivi_cvladi; }
};
//      Form1.h ფაილი
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Klasi1::statikuri_Minicheba1(ricxvi1);
Klasi1^ obj1 = gcnew Klasi1();
Klasi1::statikuri_Minicheba2(obj1, ricxvi2);

label1->Text = obj1->Naxva1().ToString();
label2->Text = obj1->Naxva2().ToString();

obj1->statikuri_Minicheba2(obj1, ricxvi2);
label3->Text = obj1->Naxva2().ToString();
}
```

კლასში გამოცხადებულია ერთი სტატიკური ცვლაი და ერთი სტატიკური ფუნქცია. ფუნქცია სტატიკურ ცვლადს ანიჭებს თავისი პარამეტრის მნიშვნელობას.

Klasi1::statikuri_Minicheba2(obj1, ricxvi2);

სტრიქონში statikuri_Minicheba2() სტატიკური ფუნქცია გამოიძახება კლასის სახელისა და ხილვადობის გაფართოების უბნის მეშვეობით,

obj1->statikuri_Minicheba2(obj1, ricxvi2);

სტრიქონში კი - obj1 ობიექტიდან.

კლასის სტატიკური წევრი არსებობს ამ კლასის ტიპის ობიექტის შექმნამდე. კლასის სტატიკური წევრების ტრადიციული გამოყენება არის დანაწილებად რესურსებთან კოორდინირებული მიმართვა, როგორიცაა დისკური მეხსიერება, პრინტერი ან ქსელური სერვერი და ა.შ.

რთული კლასი

ერთი ობიექტი შეიძლება მეორე ობიექტს შეიცავდეს. მაგალითად, ობიექტი „უნივერსიტეტი" შეიძლება შეიცავდეს „სტუდენტი" ობიექტს. ამ დროს კლასის ცვლადის გამოცხადება ხდება სხვა კლასის გამოყენებით. შედეგად, შეგვიძლია რთული კლასების შექმნა, რომლის წევრებს ექნებათ სხვა კლასების ტიპები. მაგალითი:

```
//      რთულ კლასთან მუშაობის დემონსტრირება
ref class Studenti {                    //      მარტივი კლასი
public :
```

```
static    array <System::String^>^ gvari = gcnew array <System::String^>(10);
static    array <System::String^>^ fakulteti = gcnew array <System::String^>(10);
int kursi;
void Funqcia_Studenti() {
System::Windows::Forms::MessageBox::Show
                        (L"მარტივი კლასი\n" + gvari[0] + "\n" + fakulteti[0] + "\n" + kursi.ToString());
}
};
ref class Universiteti {                    //      რთული კლასი
public : Studenti studenti;                 //      studenti ცვლადს აქვს Studenti კლასის ტიპი
static array <System::String^>^ misamarti = gcnew array <System::String^>(10);
void Funqcia_Universiteti() {
System::Windows::Forms::MessageBox::Show(L"რთული კლასი\n" + misamarti[0]);
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Universiteti^ obieqti_universiteti = gcnew Universiteti();

obieqti_universiteti->studenti.gvari[0] = textBox1->Text;
obieqti_universiteti->studenti.fakulteti[0] = textBox2->Text;
obieqti_universiteti->studenti.kursi = Convert::ToInt32(textBox3->Text);
obieqti_universiteti->studenti.Funqcia_Studenti();
obieqti_universiteti->misamarti[0] = textBox4->Text;
obieqti_universiteti->Funqcia_Universiteti();
}
```

ჩადგმული კლასი

ერთი კლასის შიგნით დასაშვებია მეორე კლასის გამოცხადება, ანუ ერთი კლასი შეიძლება ჩადგმული იყოს მეორის შიგნით. ჩადგმულ კლასს ***შიგა კლასი*** ეწოდება, მის შემცველ კლასს კი - ***გარე კლასი***. ჩადგმულ კლასი იქცევა ისე, როგორც მისი შემცველი კლასის სტატიკური წევრი. ჩადგმულ კლასი შეიძლება იყოს როგორც პრივატული, ისე ღია.

ჩადგმულ კლასს შეუძლია მიმართოს მისი შემცველი კლასის ყველა სტატიკურ წევრს. ეგზემპლარების ყველა წევრი შეიძლება იყოს მისაწვდომი შემცველი კლასის ტიპის ობიექტის, მიმთითებლის ან ასეთ ობიექტზე მიმართვის საშუალებით. შემცველი კლასიდან შესაძლებელია მიმართვა ჩადგმული კლასის მხოლოდ ღია წევრებთან. მაგრამ, ჩადგმულ კლასში, რომელიც შემცველ კლასში გამოცხადებულია როგორც private, წევრები ხშირად გამოცხადებულია როგორც public, რათა უზრუნველყოფილი იყოს თავისუფალი მიმართვა მთელ ჩადგმულ კლასთან შემცველი კლასის ფარგლებში. ჩადგმული კლასი შეგვიძლია გამოვიყენოთ მაშინ, როდესაც გვინდა შევქმნათ ტიპი, რომელიც გამოყენებული იქნება მხოლოდ სხვა ტიპის შიგნით. მოცემული პროგრამით ხდება ჩადგმულ კლასთან მუშაობის დემონსტრირება.

```
//      ჩადგმულ კლასთან მუშაობის დემონსტრირება
ref class Universiteti {                    //      გარე კლასი
public : ref class Studenti {               //      შიგა კლასი
public :
```

```
static array <System::String^>^ gvari = gcnew array <System::String^>(10);
static array <System::String^>^ fakulteti = gcnew array <System::String^>(10);
static int^ kursi;
void Funqcia_Universiteti() {
System::Windows::Forms::MessageBox::Show(L"შიგა კლასი\n" +
gvari[0] + "\n" + fakulteti[0] + "\n" + kursi->ToString());
}
};                                          //      შიგა კლასის დასასრული
Studenti^ studenti;
static array <System::String^>^ misamarti = gcnew array <System::String^>(10);
void Funqcia_Studenti() {
System::Windows::Forms::MessageBox::Show(L"გარე კლასი\n" + misamarti[0]);
}
};                                          //      გარე კლასის დასასრული
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
Universiteti^ obieqti = gcnew Universiteti();
obieqti->Funqcia_Studenti();
obieqti->misamarti[0] = textBox4->Text;
obieqti->studenti->gvari[0] = textBox1->Text;
obieqti->studenti->fakulteti[0] = textBox2->Text;
obieqti->studenti->kursi = Convert::ToInt32(textBox3->Text);
obieqti->studenti->Funqcia_Universiteti();
}
```

სტრუქტურა

სტრუქტურა არის კლასის ტიპი, რომელსაც კლასის მსგავსად აქვს საკუთარი წევრები: ცვლადები და ფუნქციები. სტრუქტურასა და კლასს შორის ერთადერთი განსხვავება იმაში მდგომარეობს, რომ ავტომატურად (გაჩუმებით) კლასის წევრები დახურულია, სტრუქტურის წევრები კი - ღია. სტრუქტურის სინტაქსია:

struct ***ტიპის_სახელი*** **{**

//   ღია ცვლადები და ფუნქციები

private :

//   დახურული ცვლადები და ფუნქციები

} ***ობიექტების_სია*****;**

როგორც ვხედავთ, სტრუქტურის გამოცხადებისათვის გამოიყენება struct საკვანძო სიტყვა. მოცემული პროგრამით ხდება სტრუქტურასთან მუშაობის დემონსტრირება.

```
//      სტრუქტურასთან მუშაობის დემონსტრირება
public ref struct Samkutxedi {
//      ცვლადების გამოცხადება
int gverdi1;
int gverdi2;
int gverdi3;
//      კონსტრუქტორის განსაზღვრა
Samkutxedi(int par1, int par2, int par3) {
```

```
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3;
}
//        ფუნქციის განსაზღვრა
int Perimetri() {
return perimetri;
}
private : int perimetri;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int g1 = Convert::ToInt32(textBox1->Text);
int g2 = Convert::ToInt32(textBox2->Text);
int g3 = Convert::ToInt32(textBox3->Text);
Samkutxedi^ struqtura1 = gcnew Samkutxedi(g1, g2, g3);
int Shedegi = struqtura1->Perimetri();
label1->Text = Shedegi.ToString();
}
```

შეიძლება ერთი სტრუქტურის მეორეში გადაწერა. მაგალითად,

```
Samkutxedi struqtura1(g1, g2, g3);
Samkutxedi struqtura2 = struqtura1;
```

ჩამოთვლა

იმ შემთხვევებში, როდესაც გვჭირდება ისეთი ცვლადები, რომლებიც მნიშვნელობებს იღებენ დასაშვები მნიშვნელობების შეზღუდული ნაკრებიდან და რომლებზეც მიმართვა მოხერხებული იქნება რაიმე ჭდის გამოყენებით, გამოიყენება ***ჩამოთვლები*** (ენუმერაციები, enumerations). ჩამოთვლის გამოცხადების სინტაქსია:

```
access enum class name {enumerator-list} var;
access enum struct name {enumerator-list} var;
```

ჩამოთვლის გამოცხადების მაგალითი:

```
enum class KvirisDge { Orshabati, Samshabati, Otxshabati, Xutshabati, Paraskevi, Shabati, Kvira };
```

აქ განისაზღვრება ჩამოთვლის ტიპი - KvirisDge და ამ ტიპის ცვლადს შეიძლება მივანიჭოთ ჩამოთვლით განსაზღვრული სიდიდეებიდან ერთ-ერთი: Orshabati, Samshabati და ა.შ.

ჩამოთვლაში მივმართავთ მუდმივებს ჩამოთვლის ტიპის სახელის საშუალებით:

```
KvirisDge kd = KvirisDge::Samshabati;
```

ჩამოთვლაში განსაზღვრული მუდმივები წარმოადგენენ ობიექტებს. ავტომატურად ისინი არიან Int32 ტიპის ობიექტები. ამიტომ, შეგვიძლია მუდმივა int ტიპად გარდავქმნათ. მაგალითი:

```
//        ჩამოთვლასთან მუშაობის დემონსტრირება
enum class KvirisDge { Orshabati, Samshabati, Otxshabati, Xutshabati, Paraskevi, Shabati, Kvira };
//
private: System::Void button15_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
KvirisDge kd = KvirisDge::Samshabati;
label1->Text = (KvirisDge::Paraskevi).ToString();
label2->Text = safe_cast<int>(KvirisDge::Paraskevi).ToString();
}
```

რადგან, ჩამოთვლა არის კლასის ტიპის მქონე, მას ვერ განვსაზღვრავთ ლოკალურად ფუნქციის შიგნით. ის უნდა განისაზღვროს როგორც გლობალური, ფუნქციის გარეთ. ჩამოთვლაში მუდმივებს შეიძლება ჰქონდეს 5.1 ცხრილში მოცემული ტიპები.

ცხრილი 5.1. ჩამოთვლის მუდმივების ტიპები

საბაზო ტიპი	ზომა ბაიტებში	CLI Value Class
bool	1	System::Boolean
char	1	System::SByte
signed char	1	System:: SByte
unsigned char	1	System:: Byte
short	2	System::Int16
unsigned short	2	System:: UInt16
int	4	System::Int32
unsigned int	4	System::UInt32
long	4	System:: Int32
unsigned long	4	System:: UInt32
long long	8	System::Int64
unsigned long long	8	System::UInt64

ჩამოთვლის მუდმივებისთვის შეგვიძლია ტიპის განსაზღვრა. მოცემულ მაგალითში მუდმივებს Char ტიპი აქვს:

```
enum class Tveebi : Char { Ianvari, Tebervali, Marti, Aprili, Maisi, Ivnisi,
                           Ivlisi, Agvisto, Seqtemberi, Oqtomberi, Noemberi, Dekemberi };
```

პირველი მუდმივას სიდიდე ავტომატურად იქნება 0.

შეგვიძლია აშკარად მივუთითოთ სიდიდე რომელიმე მუდმივასთვის:

```
enum class Tveebi : Char { Ianvari = 2, Tebervali, Marti, Aprili, Maisi, Ivnisi,
                           Ivlisi, Agvisto, Seqtemberi, Oqtomberi, Noemberi, Dekemberi };
```

შედეგად, Tebervali მუდმივას სიდიდე იქნება 3 და ა.შ.

კიდევ ერთი მაგალითი:

```
enum class Tveebi : Char { Ianvari = 20, Tebervali = 10, Marti, Aprili, Maisi, Ivnisi,
                           Ivlisi, Agvisto, Seqtemberi, Oqtomberi, Noemberi, Dekemberi };
```

შედეგად, Marti მუდმივას სიდიდე იქნება 11 და ა.შ.

მუდმივებს შეიძლება ჰქონდეთ ერთნაირი მნიშვნელობები:

```
enum class Tveebi : bool { Ianvari = true, Tebervali = true, Marti = true, Aprili = true, Maisi = true,
                           Ivnisi = true, Ivlisi = false, Agvisto = false, Seqtemberi = false,
                           Oqtomberi = false, Noemberi = false, Dekemberi = false };
```

გაერთიანება

გაერთიანება (union) არის კლასის ტიპი, რომელსაც კლასის მსგავსად აქვს საკუთარი წევრები: ცვლადები და ფუნქციები, კონსტრუქტორები და დესტრუქტორები. გაერთიანებაში

სტრუქტურის მსგავსად, ყველა წევრი ღიაა მანამ, სანამ არ იქნება მითითებული private: მოდიფიკატორი. გაერთიანების ყველა მონაცემი მოთავსებულია ოპერატიული მეხსიერების ერთსა და იმავე უბანში.

გაერთიანების გამოყენებას შემდეგი შეზღუდვები აქვს:

- გაერთიანება არ შეიძლება იყოს არც წინაპარი და არც საბაზო კლასი;
- გაერთიანებას არ შეიძლება ჰქონდეს სტატიკური წევრი;

მოცემული პროგრამით ხდება გაერთიანებასთან მუშაობის დემონსტრირება.

```
//      გაერთიანებასთან მუშაობის დემონსტრირება
union Gaertianeba {
        long long mteli;
        double wiladi;
        bool logikuri;
        System::Char simbolo;
        Gaertianeba(long long par1, double par2, bool par3, System::Char par4) {
        mteli = par1;
        wiladi = par2;
        logikuri = par3;
        simbolo = par4;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
long long  ricxvi1 = Convert::ToInt64(textBox1->Text);
double ricxvi2 = Convert::ToDouble(textBox2->Text);
bool cvladi3 = Convert::ToBoolean(textBox3->Text);
Char cvladi4 = Convert::ToChar(textBox4->Text[0]);
Gaertianeba obj1(ricxvi1, ricxvi2, cvladi3, cvladi4);
label1->Text = obj1.mteli.ToString() + "  " + obj1.wiladi.ToString() + "   " +
                obj1.logikuri.ToString() + "   " + obj1.simbolo;
}
```

ლიტერალური ველი

C++/CLI ენაში განსაზღვრულია ***ლიტერალური ველები***, რომლებიც განკუთვნილია კლასში რიცხვითი მუდმივების წარმოსადგენად. ამისათვის, გამოიყენება literal საკვანძო სიტყვა. მოცემული პროგრამით ხდება ლიტერალურ ველებთან მუშაობის დემონსტრირება.

```
//      ლიტერალურ ველებთან მუშაობის დემონსტრირება
ref class Samkutxedi_1 {
private :
int gverdi1;
int gverdi2;
literal int gverdi3 = 10;                   //      ლიტერალის გამოცხადება
int perimetri;
double fartobi;
public :
Samkutxedi_1(int par1, int par2) {
```

```
gverdi1 = par1;
gverdi2 = par2;
perimetri = gverdi1 + gverdi2 + gverdi3 ;
}
int NaxvaPerimetri() {
return perimetri;
}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Samkutxedi_1^ obj1= gcnew Samkutxedi_1(cvladi1, cvladi2);
label3->Text = L"სამკუთხედის პერიმეტრი = " + obj1->NaxvaPerimetri().ToString();
}
```

ერთი ლიტერალური ველის მნიშვნელობა შეგვიძლია განვსაზღვროთ სხვა ლიტერალური ველების საშუალებით:

```
//      ერთი ლიტერალური ველის მნიშვნელობის
//      განსაზღვრა სხვა ლიტერალური ველებით
ref class Samkutxedi_2 {
private :
literal int gverdi1 = 10;
literal int gverdi2 = 20;
int gverdi3;
int perimetri;
//      fartobi ლიტერალური ველის მნიშვნელობა განისაზღვრება
//      gverdi1 და gverdi2 ლიტერალური ველებით
literal double fartobi = ( gverdi1 * gverdi2 ) / 2;
public :
Samkutxedi_2(int par3) {
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3 ;
}
int NaxvaPerimetri() {
return perimetri;
}
double Naxva_Fartobi() {
return fartobi;
}
};
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
Samkutxedi_2^ obj1= gcnew Samkutxedi_2(cvladi1);
label1->Text = L"სამკუთხედის პერიმეტრი = " + obj1->NaxvaPerimetri().ToString();
label2->Text = L"სამკუთხედის ფართობი = " + obj1->Naxva_Fartobi().ToString();
}
```

initonly ველი

ლიტერალური ველების ინიციალიზება კონსტრუქტორში არ შეიძლება. ამ პრობლემის გადასაჭრელად C++/CLI ენაში შემოტანილია **initonly *ველი***. ის არის მუდმივა, რომელიც ინიციალიზებული შეიძლება იყოს კონსტრუქტორში. მოცემული პროგრამით ხდება არასტატიკურ initonly ველთან მუშაობის დემონსტრირება.

```
//      არასტატიკურ initonly ველთან მუშაობა
ref class Samkutxedi {
private :
        int gverdi1;
        int gverdi2;
        int gverdi3;
public :
        initonly int perimetri;
        Samkutxedi(int par1, int par2, int par3) : gverdi1(par1), gverdi2(par2), gverdi3(par3) {
        perimetri = gverdi1 + gverdi2 + gverdi3;
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi^ obj1 = gcnew Samkutxedi(ricxvi1, ricxvi2, ricxvi3);
label1->Text = obj1->perimetri.ToString();
}
```

initonly ველი შეიძლება იყოს სტატიკური. მოცემული პროგრამაში ხდება სტატიკურ initonly ველთან მუშაობის დემონსტრირება.

```
//      სტატიკურ initonly ველთან მუშაობა
ref class Samkutxedi {
private :
        int gverdi1;
        int gverdi2;
public :
        int perimetri;
        initonly static int gverdi3 = 50;
        Samkutxedi(int par1, int par2) : gverdi1(par1), gverdi2(par2) {
                perimetri = gverdi1 + gverdi2 + gverdi3;
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Samkutxedi^ obj1 = gcnew Samkutxedi(ricxvi1, ricxvi2);
```

```
label1->Text = obj1->perimetri.ToString();
}
```

კონსტრუქტორში არ შეიძლება სტატიკური ველებისთვის მნიშვნელობების მინიჭება ან შეცვლა. საქმე ის არის, რომ სტატიკური ველები დანაწილებულია კლასის ყველა ობიექტს შორის და მათი შეცვლა კონსტრუქტორის ყოველი გამოძახებისას დაუშვებელია.

სტატიკური კონსტრუქტორი

სტატიკურია კონსტრუქტორი, რომელიც გამოცხადებულია static საკვანძო სიტყვის გამოყენებით. ის გამოიყენება ჩვეულებრივი და initonly სტატიკური ველების ინიციალიზებისთვის. სტატიკურ კონსტრუქტორს არ აქვს პარამეტრები და ინიციალიზების სია. ის პრივატულია ყოველთვის, იმ შემთხვევაშიც კი, თუ მოთავსებულია public განყოფილებაში. სტატიკური კონსტრუქტორი შეიძლება განვსაზღვროთ როგორც მნიშვნელობითი კლასებისთვის, ისე მიმართვითი კლასებისთვის. ჩვენ არ შეგვიძლია უშუალოდ გამოვიძახოთ სტატიკური კონსტრუქტორი. ის ავტომატურად გამოიძახება ჩვეულებრივი კონსტრუქტორის პირველ შესრულებამდე. სტატიკური ველები, რომლებსაც მინიჭებული აქვთ საწყისი მნიშვნელობები, ინიციალიზებული იქნება სტატიკური კონსტრუქტორის შესრულებამდე. მოცემული პროგრამით ხდება სტატიკურ კონსტრუქტორთან მუშაობის დემონსტრირება.

```
//        სტატიკურ კონსტრუქტორთან მუშაობა
ref class Samkutxedi_1 {
private :
        int gverdi1;
        int gverdi2;
        //        სტატიკური კონსტრუქტორი
        static Samkutxedi_1() { gverdi3 = 50; }
public :
        int perimetri;
        initonly static int gverdi3 = 40;
        Samkutxedi_1(int par1, int par2) : gverdi1(par1), gverdi2(par2) {
                perimetri = gverdi1 + gverdi2 + gverdi3;
        }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
 Samkutxedi_1^ obj1 = gcnew Samkutxedi_1(15, 90);
 label1->Text = obj1->perimetri.ToString();                        //        perimetri = 15 + 90 + 50
}
```

        აქ gverdi3 ცვლადის მნიშვნელობა იქნება 50 და არა 40.

შემთხვევითი რიცხვების გენერატორი

C++ ენაში განსაზღვრულია System::Random კლასი შემთხვევით რიცხვებთან სამუშაოდ. 5.2 ცხრილში მოცემულია ამ კლასის ფუნქციები, რომლებიც ახდენენ შემთხვევითი რიცხვების გენერირებას.

მოვიყვანოთ ორი პროგრამა, რომლებშიც შემთხვევითი რიცხვების გენერატორი გამოიყენება მასივების შესავსებად. ამ პროგრამით ორგანზომილებიანი მასივის შესავსებად გამოიყენება Next()ფუნქცია.

ცხრილი 5.2. Random კლასის ფუნქციები

ფუნქცია	აღწერა
int Random.Next(int ცვლადი)	გასცემს არაუარყოფით შემთხვევით მთელ რიცხვს, რომელიც ნაკლებია მითითებული ცვლადის მნიშვნელობაზე
double Random.NextDouble()	გასცემს შემთხვევით წილად რიცხვს, რომლის მნიშვნელობა მოთავსებულია 0.0-სა და 1.0-ს შორის
void Random.NextBytes(byte[ ] მასივი)	მითითებულ მასივს შეავსებს შემთხვევითი რიცხვებით, რომელთა ტიპია byte

```
{
//      შემთხვევითი რიცხვების გენერატორი გამოიყენება
//      ორგანზომილებიანი მასივის შესავსებად
private: System::Void button13_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text="";
int max = Convert::ToInt32(textBox1->Text);
array <int, 2>^ mas = gcnew array <int, 2>(3,3);
System::Random obieqti;

for ( int striqoni = 0; striqoni < 3; striqoni++ )
        for ( int sveti = 0; sveti < 3; sveti++ )
                mas[striqoni, sveti] = obieqti.Next(max);

for ( int striqoni = 0; striqoni < 3; striqoni++ )
{
        for ( int sveti = 0; sveti < 3; sveti++ )
                label1->Text += mas[striqoni, sveti].ToString() + "   ";
        label1->Text = label1->Text + "\n";
}
}
```

ამ პროგრამით ერთგანზომილებიანი მასივის შესავსებად გამოიყენება NextBytes() ფუნქცია.

```
{
//      შემთხვევითი რიცხვების გენერატორის გამოყენება
//      ერთგანზომილებიანი მასივის შესავსებად
private: System::Void button14_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text="";
Random^ obj = gcnew Random;
array<Byte>^ masivi = gcnew array<Byte>(5);
//      masivi  მასივის შევსება შემთხვევითი რიცხვებით
obj->NextBytes(masivi);
for ( int i = 0; i < 5; i++ )
        label1->Text += masivi[i].ToString() + "   ";      }
```

თავი 7. ფუნქციები უფრო დაწვრილებით. პოლიმორფიზმი

ფუნქციისათვის არგუმენტის გადაცემის ხერხები

ფუნქციას არგუმენტები შეგვიძლია ორი გზით გადავცეთ. პირველია, არგუმენტის გადაცემა მნიშვნელობის მიხედვით. ამ შემთხვევაში, არგუმენტის მნიშვნელობის ასლი ენიჭება ფუნქციის პარამეტრს. თუმცა, პარამეტრისა და არგუმენტის მნიშვნელობები მეხსიერების სხვადასხვა უბანშია მოთავსებული. შედეგად, პარამეტრის მნიშვნელობის ცვლილება არ შეცვლის არგუმენტის მნიშვნელობას. მეორეა, არგუმენტის გადაცემა მიმართვის მიხედვით. ამ შემთხვევაში, ფუნქციის პარამეტრს გადაეცემა არგუმენტზე მიმართვა (არგუმენტის მისამართი). შედეგად, პარამეტრის მნიშვნელობის შეცვლისას, შესაბამისად შეიცვლება არგუმენტის მნიშვნელობაც. ეს იმიტომ ხდება, რომ პარამეტრიც და არგუმენტიც მეხსიერების ერთსა და იმავე უბანს მიმართავს, ანუ პარამეტრი და არგუმენტი მეხსიერების ერთსა და იმავე უბანშია მოთავსებული. განვიხილოთ არგუმენტის გადაცემის თითოეული გზის რეალიზების მაგალითები.

მოცემული მშობლიური C++ პროგრამით ხდება ფუნქციისთვის არგუმენტის გადაცემა მნიშვნელობის მიხედვით.

```
//      ფუნქციისათვის არგუმენტის გადაცემა მნიშვნელობის მიხედვით
class Chveulebrivi_Tipi {
//      ამ ფუნქციის შესრულება არ გამოიწვევს არგუმენტის შეცვლას
public : void Funqcia(int par1, int par2) {
        par1 = par1 + par2;
        par2 = - par2;
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
Chveulebrivi_Tipi obieqti;
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე:  " +
                ricxvi1.ToString() + "   " + ricxvi2.ToString();
obieqti.Funqcia(ricxvi1, ricxvi2);              //      Funqcia ფუნქციის გამოძახება
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
                ricxvi1.ToString() + "   " + ricxvi2.ToString();
}
```

იგივე პროგრამის C++/CLI ვერსია:

```
ref class Chveulebrivi_Tipi {
//      ამ ფუნქციის შესრულება არ გამოიწვევს არგუმენტის შეცვლას
public : void Funqcia(int par1, int par2) {
        par1 = par1 + par2;
        par2 = - par2;
}
};
```

```
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Chveulebrivi_Tipi^ obieqti = gcnew Chveulebrivi_Tipi();
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე:  " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
obieqti->Funqcia(ricxvi1, ricxvi2);                    //          Funqcia ფუნქციის გამოძახება
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
}
```

მოცემული მშობლიური C++ პროგრამით ხდება ფუნქციისთვის არგუმენტის გადაცემა მიმართვის მიხედვით.

```
//          ფუნქციისათვის არგუმენტის გადაცემა მნიშვნელობის მიხედვით
class Mimartviti_Tipi {
//          ამ ფუნქციის შესრულება არ გამოიწვევს არგუმენტის შეცვლას
public : void Funqcia(int &par1, int &par2) {
          par1 = par1 + par2;
          par2 = - par2;
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
Mimartviti_Tipi *obieqti = new Mimartviti_Tipi;
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე:  " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
obieqti->Funqcia(ricxvi1, ricxvi2);                    //          Funqcia ფუნქციის გამოძახება
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
}
```

იმავე პროგრამის C++/CLI ვერსია:

```
ref class Mimartviti_Tipi {
//          ამ ფუნქციის შესრულება არ გამოიწვევს არგუმენტის შეცვლას
public : void Funqcia(int &par1, int &par2) {
          par1 = par1 + par2;
          par2 = - par2;
}
};
//
```

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Mimartviti_Tipi^ obieqti = gcnew Mimartviti_Tipi;
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე:  " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
obieqti->Funqcia(ricxvi1, ricxvi2);                    //          Funqcia ფუნქციის გამოძახება
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
ricxvi1.ToString() + "   " + ricxvi2.ToString();
}
```

ფუნქციისთვის ობიექტის გადაცემა

ფუნქციას პარამეტრად ჩვეულებრივი ტიპების მნიშვნელობების გარდა შეგვიძლია გადავცეთ ობიექტიც. C++ ენაში, ფუნქციას ობიექტი მნიშვნელობის მიხედვით გადაეცემა. ფუნქციაში იქმნება არგუმენტის (ობიექტის) ასლი, რომელთანაც ფუნქცია მუშაობს. ამიტომ, ობიექტის ასლში შეტანილი ცვლილებები, ობიექტს არ შეეხება. მაგალითი:

```
//          ფუნქციისთვის ობიექტის გადაცემის დემონსტრირება
class Klasi1 {
public : int cvladi1, cvladi2;
Klasi1(int par1, int par2) {
        cvladi1 = par1;
        cvladi2 = par2;
}
//
void shecvla(Klasi1 obieqti1) {
        obieqti1.cvladi1 = obieqti1.cvladi1 + obieqti1.cvladi2;
        obieqti1.cvladi2 = -obieqti1.cvladi2;
}
};
//
private: System::Void button7_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

Klasi1 obieqti2(ricxvi1, ricxvi2);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე: " +
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
obieqti2.shecvla(obieqti2);
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
```

```
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
}
```

იმავე პროგრამის C++/CLI ვერსია:

```
value class Klasi1 {
public : int cvladi1, cvladi2;
Klasi1(int par1, int par2) {
        cvladi1 = par1;
        cvladi2 = par2;
}
//
void shecvla(Klasi1 obieqti1) {
        obieqti1.cvladi1 = obieqti1.cvladi1 + obieqti1.cvladi2;
        obieqti1.cvladi2 = -obieqti1.cvladi2;
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

Klasi1 obieqti2(ricxvi1, ricxvi2);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე: " +
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
obieqti2.shecvla(obieqti2);
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
}
```

მოცემული მშობლიური C++ პროგრამით ხდება მიმთითებლის გამოყენება ფუნქციისთვის მიმართვის მიხედვით ობიექტის (არგუმენტის) გადასაცემად.

```
//      მიმთითებლის გამოყენებით ფუნქციისთვის
//      ობიექტის გადაცემა მიმართვის მიხედვით
class Mimartviti_tipi {
public : int cvladi1, cvladi2;
Mimartviti_tipi(int par1, int par2) {
        cvladi1 = par1;
        cvladi2 = par2;
}
//
void shecvla(Mimartviti_tipi *obieqti1) {
        obieqti1->cvladi1 = obieqti1->cvladi1 + obieqti1->cvladi2;
        obieqti1->cvladi2 = -obieqti1->cvladi2;
}
};
//
```

```
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

Mimartviti_tipi obieqti2(ricxvi1, ricxvi2);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე: " +
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
obieqti2.shecvla(&obieqti2);
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
obieqti2.cvladi1.ToString() + "   " + obieqti2.cvladi2.ToString();
}
```

პროგრამის obieqti2.shecvla(&obieqti2) სტრიქონში ხდება obieqti2 ობიექტის shecvla ფუნქციის გამოძახება. მისი არგუმენტია &obieqti2. ამიტომ, obieqti2-ის მიმართვა ენიჭება obieqti1 ცვლადს. შედეგად, obieqti1 და obieqti2 ობიექტები მეხსიერების ერთსა და იმავე უბანს მიმართავენ. ამიტომ, obieqti1 ობიექტის ცვლადების ცვლილება ავტომატურად გამოიწვევს obieqti2 ობიექტის ცვლადების შეცვლას.

იმავე პროგრამის C++/CLI ვერსია:

```
ref class Mimartviti_tipi {
public : int cvladi1, cvladi2;
Mimartviti_tipi(int par1, int par2) {
        cvladi1 = par1;
        cvladi2 = par2;
}
//
void shecvla(Mimartviti_tipi^ obieqti1) {
        obieqti1->cvladi1 = obieqti1->cvladi1 + obieqti1->cvladi2;
        obieqti1->cvladi2 = -obieqti1->cvladi2;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1, ricxvi2;

ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);

Mimartviti_tipi^ obieqti2 = gcnew Mimartviti_tipi(ricxvi1, ricxvi2);

label1->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახებამდე: " +
                obieqti2->cvladi1.ToString() + "   " + obieqti2->cvladi2.ToString();
obieqti2->shecvla(obieqti2);
label2->Text = L"არგუმენტების მნიშვნელობები ფუნქციის გამოძახების შემდეგ: " +
                obieqti2->cvladi1.ToString() + "   " + obieqti2->cvladi2.ToString();
}
```

მოცემული მშობლიური C++ პროგრამით ხდება სამკუთხედების (ობიექტების) შედარება. მათი ზომები ინახება Samkutxedi კლასის ობიექტებში.

```
//        ფუნქციისათვის ობიექტის გადაცემის დემონსტრირება
class Samkutxedi {
int gverdi1, gverdi2, gverdi3;
int perimetri;
public : Samkutxedi(int par1, int par2, int par3) {
        gverdi1  = par1;
        gverdi2  = par2;
        gverdi3  = par3;
        perimetri = gverdi1 + gverdi2 + gverdi3;
}
/*
IgiveSamkutxedi() ფუნქცია გასცემს true მნიშვნელობას, თუ Sam1 ობიექტში არის სამკუთხედის
ზომების იგივე  მნიშვნელობები, რაც გამომძახებელ ობიექტში
*/
bool IgiveSamkutxedi(Samkutxedi Sam1) {
if (  ( Sam1.gverdi1 == gverdi1 ) && ( Sam1.gverdi2 == gverdi2 ) &&
     ( Sam1.gverdi3 == gverdi3 ) ) return true;
        else return false;
}
/*
IgivePerimetri() ფუნქცია გასცემს true მნიშვნელობას, თუ Sam1 ობიექტში არის სამკუთხედის
პერიმეტრის იგივე  მნიშვნელობა, რაც გამომძახებელ ობიექტში
*/
bool IgivePerimetri(Samkutxedi Sam1) {
if ( Sam1.perimetri == perimetri ) return true;
        else return false;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int arg1, arg2, arg3;

arg1 = Convert::ToInt32(textBox1->Text);
arg2 = Convert::ToInt32(textBox2->Text);
arg3 = Convert::ToInt32(textBox3->Text);

Samkutxedi obieqti1(arg1, arg2, arg3);
Samkutxedi obieqti2(10, 2, 5);
Samkutxedi obieqti3(Convert::ToInt32(textBox1->Text),
                        Convert::ToInt32(textBox2->Text), Convert::ToInt32(textBox3->Text));

label1->Text = L"პირველი და მეორე სამკუთხედების ზომები ერთნაირია --- "    +
obieqti1.IgiveSamkutxedi(obieqti2).ToString();
label2->Text = L"პირველი და მესამე სამკუთხედების ზომები ერთნაირია --- "   +
obieqti1.IgiveSamkutxedi(obieqti3).ToString();
```

```
label3->Text = L"მეორე და მესამე სამკუთხედების პერიმეტრები ერთნაირია --- " +
obieqti2.IgivePerimetri(obieqti3).ToString();
}
```

ამ პროგრამის Samkutxedi კლასში გამოცხადებულია IgiveSamkutxedi() ფუნქცია. მისი Sam1 პარამეტრის ტიპია IgiveSamkutxedi. ფუნქციის (Sam1.gverdi1 == gverdi1) გამოსახულებაში ხდება Sam1 ობიექტის gverdi1 ცვლადის შედარება ტოლობაზე გამომძახებელი ობიექტის gverdi1 ცვლადთან. ძირითად პროგრამაში ამ ფუნქციის გამოძახება ხდება obieqti1.IgiveSamkutxedi(obieqti2) სტრიქონში. obieqti1 არის გამომძახებელი ობიექტი. obieqti2-ის მიმართვა მიენიჭება Sam1 პარამეტრს, ამიტომ, Sam1.gverdi1 ჩანაწერში gverdi1 ეკუთვნის obieqti2-ს, შედარების (==) მარჯვნივ მითითებული gverdi1 ცვლადი კი - obieqti1-ს ანუ გამომ-ძახებელ ობიექტს.

იმავე პროგრამის C++/CLI ვარიანტი:

```
ref class Samkutxedi {
int gverdi1, gverdi2, gverdi3;
int perimetri;
public : Samkutxedi(int par1, int par2, int par3) {
        gverdi1  = par1;
        gverdi2  = par2;
        gverdi3  = par3;
        perimetri = gverdi1 + gverdi2 + gverdi3;
}
/*
IgiveSamkutxedi() ფუნქცია გასცემს true მნიშვნელობას, თუ Sam1 ობიექტში არის სამკუთხედის
ზომების იგივე  მნიშვნელობები, რაც გამომძახებელ ობიექტში
*/
bool IgiveSamkutxedi(Samkutxedi^ Sam1) {
if (  ( Sam1->gverdi1 == gverdi1 ) && ( Sam1->gverdi2 == gverdi2 ) &&
     ( Sam1->gverdi3 == gverdi3 ) ) return true;
        else return false;
}
/*
IgivePerimetri() ფუნქცია გასცემს true მნიშვნელობას, თუ Sam1 ობიექტში არის სამკუთხედის
პერიმეტრის იგივე  მნიშვნელობა, რაც გამომძახებელ ობიექტში
*/
bool IgivePerimetri(Samkutxedi^ Sam1) {
if ( Sam1->perimetri == perimetri ) return true;
        else return false;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int arg1, arg2, arg3;

arg1 = Convert::ToInt32(textBox1->Text);
arg2 = Convert::ToInt32(textBox2->Text);
arg3 = Convert::ToInt32(textBox3->Text);

Samkutxedi^ obieqti1 = gcnew Samkutxedi(arg1, arg2, arg3);
```

```
Samkutxedi^ obieqti2 = gcnew Samkutxedi(10, 2, 5);
Samkutxedi^ obieqti3 = gcnew Samkutxedi(Convert::ToInt32(textBox1->Text),
Convert::ToInt32(textBox2->Text), Convert::ToInt32(textBox3->Text));

label1->Text = L"პირველი და მეორე სამკუთხედების ზომები ერთნაირია --- "    +
obieqti1->IgiveSamkutxedi(obieqti2).ToString();
label2->Text = L"პირველი და მესამე სამკუთხედების ზომები ერთნაირია --- "   +
obieqti1->IgiveSamkutxedi(obieqti3).ToString();
label3->Text = L"მეორე და მესამე სამკუთხედების პერიმეტრები ერთნაირია --- " +
obieqti2->IgivePerimetri(obieqti3).ToString();
}
```

როგორც აღვნიშნეთ, ფუნქციას გადაეცემა არგუმენტის ასლი. როდესაც ფუნქცია მუშაობას ამთავრებს, მეხსიერებიდან ეს ასლი წაიშლება. ობიექტის ასლის შექმნისას კონსტრუქტორი არ გამოიძახება, მისი წაშლისას კი - გამოიძახება დესტრუქტორი. მაგალითი:

```
//          ფუნქციისათვის ობიექტის გადაცემის დემონსტრირება
class Obieqti_Asli {
int ricxvi;
public :
Obieqti_Asli(int par1) {
        ricxvi = par1;
        System::Windows::Forms::MessageBox::Show(L"მუშაობს კონსტრუქტორი");
}
~Obieqti_Asli() {
        System::Windows::Forms::MessageBox::Show(L"მუშაობს დესტრუქტორი");
}
int get_ricxvi() {
return ricxvi;
}
};
int Funqcia2(Obieqti_Asli obj2) {
        return obj2.get_ricxvi() + obj2.get_ricxvi();
}
//
private: System::Void button8_Click(System::Object^  sender, System::EventArgs^  e) {
Obieqti_Asli obj1(25);
Funqcia2(obj1);
}
```

პროგრამის შესრულებისას კონსტრუქტორი ერთხელ გამოიძახება, დესტრუქტორი კი - ორჯერ. დესტრუქტორი პირველად გამოიძახება მაშინ, როდესაც ობიექტის ასლი იშლება, მეორედ კი მაშინ, როდესაც თვით ობიექტი იშლება.

იმავე პროგრამის C++/CLI ვერსია:

```
ref class Obieqti_Asli {
int ricxvi;
public :
Obieqti_Asli(int par1) {
        ricxvi = par1;
        System::Windows::Forms::MessageBox::Show(L"მუშაობს კონსტრუქტორი");
```

```
}
~Obieqti_Asli() {
	System::Windows::Forms::MessageBox::Show(L"მუშაობს დესტრუქტორი");
}
int get_ricxvi() {
return ricxvi;
}
};
int Funqcia2(Obieqti_Asli^ obj2) {
	return obj2->get_ricxvi() + obj2->get_ricxvi();
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Obieqti_Asli^ obj1 = gcnew Obieqti_Asli(25);
Funqcia2(obj1);
delete obj1;
}
```

ვინაიდან, იმ ობიექტის დესტრუქტორი, რომელიც ფუნქციისათვის გადასაცემი არგუმენტის ასლია, გამოიძახება ფუნქციის მუშაობის დამთავრების შემდეგ, შესაძლებელია გაჩნდეს პრობლემები. კერძოდ, თუ ობიექტისთვის, რომელიც არგუმენტად გამოიყენება, გამოყოფილია დინამიკური მეხსიერება, რომელიც თავისუფლდება მისი წაშლისას, მაშინ ამ ობიექტის ასლისთვისაც დესტრუქტორის გამოძახებისას გათავისუფლდება იგივე მეხსიერება. ეს კი გამოიწვევს საწყისი ობიექტის დაზიანებას. ასეთი სახის შეცდომის თავიდან ასაცილებლად, უნდა დავრწმუნდეთ, რომ იმ ობიექტის ასლის დესტრუქტორი, რომელიც არგუმენტად გამოიყენება, არ იმოქმედებს საწყის ობიექტზე.

დესტრუქტორით ჩვენთვის საჭირო მონაცემების წაშლის პრობლემის თავიდან აცილების ერთ-ერთი გზაა ფუნქციისთვის არა არგუმენტის, არამედ მისი მისამართის გადაცემა. თუ ფუნქციას ობიექტის მისამართი გადაეცემა, მაშინ ახალი ობიექტი არ იქმნება და ამიტომ, ფუნქციის მიერ მნიშვნელობის დაბრუნებისას დესტრუქტორი არ გამოიძახება. ამ პრობლემის უფრო უკეთესი გადაწყვეტაა ***ასლის კონსტრუქტორის*** (*copy constructor*) გამოყენება.

ფუნქციის მიერ ობიექტის დაბრუნება

ფუნქციას შეუძლია დააბრუნოს როგორც ნებისმიერი ტიპის მონაცემი, ისე კლასის ობიექტიც. მაგალითად განვიხილოთ ორი პროგრამა. პირველ პროგრამაში ფუნქცია გასცემს სტრიქონების მასივის ელემენტს, მეორეში კი - ჩვენ მიერ შექმნილი კლასის ობიექტს.

```
// ფუნქციის მიერ სტრიქონის დაბრუნების დემონსტრირება
ref class ChemiKlasi {
static array<System::String^>^ xili = gcnew array<System::String^>(4)
                                        { L"ვაშლი", L"მსხალი", L"ლეღვი", L"ყურძენი" };
public : System::String^ StriqonisDabruneba(int indexi) {
	if ( ( indexi >= 0 ) && ( indexi < xili->Length ) ) return  xili[indexi];
	else return L"შეტანილია არასწორი ინდექსი";
}
};
//
```

```
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
//        ობიექტის დაბრუნების დემონსტრირება
ChemiKlasi obieqti;

label1->Text = obieqti.StriqonisDabruneba(3);
label2->Text = obieqti.StriqonisDabruneba(15);
}
```

პროგრამით ხდება ობიექტის დაბრუნების დემონსტრირება.

```
//        ფუნქციის მიერ ობიექტის დაბრუნების დემონსტრირება
ref class ChemiKlasi {
        static    array<int>^ mas = gcnew array<int>(5);
public :
        void Naxva(System::Windows::Forms::Label^ lab1) {
                for ( int indexi = 0; indexi < 5; indexi++ )
                        lab1->Text += mas[indexi].ToString() + "  ";
        }
        void Minicheba(array<int>^ par) {
                for ( int indexi = 0; indexi < 5; indexi++ )
                        mas[indexi] = indexi + 5;
        }
//        ფუნქციის გამოცხადება
ChemiKlasi^ Funqcia(array<int>^ par1) {
ChemiKlasi^ obj1;
obj1->Minicheba(par1);
return obj1;
}
};
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
//         ობიექტის დაბრუნების დემონსტრირება
label1->Text = "";
ChemiKlasi^ obieqti1;
ChemiKlasi^ obieqti2;
array<int>^ masivi = gcnew array<int> {1, 2, 3, 4, 5};
obieqti1 = obieqti2->Funqcia(masivi);
obieqti1->Naxva(label1);
}
```

ChemiKlasi კლასში გამოცხადებული Funqcia() ფუნქცია გასცემს ChemiKlasi ტიპის ობიექტს. ძირითად პროგრამაში ეს გაცემული ობიექტი ენიჭება obieqti1 ობიექტს.

ფუნქციისთვის ცვლადი რაოდენობის არგუმენტების გადაცემა

C++/CLI გარემო უზრუნველყოფს ცვლადი რაოდენობის არგუმენტების გადაცემას ფუნქციისთვის. არგუმენტები გადაიცემა როგორც მასივი, რომელსაც წინ უძღვის მრავალწერტილი (...). მაგალითი:

```
int Funqcia(... array<int>^ masivi)
```

```
{
//        ფუნქციის კოდი
}
```

Funqcia() ფუნქცია იღებს ნებისმიერი რაოდენობის int ტიპის არგუმენტს. არგუმენტების დამუშავების მიზნით უნდა მივმართოთ masivi მასივის ელემენტებს. მოცემული პროგრამით ხდება ცვლადი რაოდენობის არგუმენტებთან მუშაობის დემონსტრირება.

```
//        ცვლადი რაოდენობის არგუმენტებთან მუშაობის დემონსტრირება
ref class Klasi {
public :
int maxSidide(...array<int>^ masivi)
{
int max;
if ( masivi->Length == 0 )
{
//  რადგან არგუმენტი არ არის მითითებული, ამიტომ ფუნქცია აბრუნებს ნულს
        return 0;
}
max = masivi[0];
for ( int indexi = 1; indexi < masivi->Length; indexi++ )
        if ( masivi[indexi] > max ) max = masivi[indexi];
return max;
}
};
//
private: System::Void button11_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj = gcnew Klasi;
int ricxvi1 = 1, ricxvi2 = 2, ricxvi3 = 3;
array<int>^ masivi1 = gcnew array<int> {10, 21, 3, 42, 5};
array<int>^ masivi2 = gcnew array<int> {10, 21, 3, 42, 5, 50, 30};

int maqsimumi1 = obj->maxSidide(masivi1);   //        ფუნქციას გადაეცემა 5-არგუმენტიანი მასივი
label1->Text = maqsimumi1.ToString();

int maqsimumi2 = obj->maxSidide(masivi2);   //        ფუნქციას გადაეცემა 7-არგუმენტიანი მასივი
label2->Text = maqsimumi2.ToString();

int maqsimumi3 = obj->maxSidide(ricxvi1, ricxvi2, ricxvi3);   //        ფუნქციას გადაეცემა 3 ცვლადი
label3->Text = maqsimumi3.ToString();

int maqsimumi4 = obj->maxSidide(ricxvi1, ricxvi2);                //        ფუნქციას გადაეცემა 2 ცვლადი
label4->Text = maqsimumi4.ToString();
}
```

ცვლადი რაოდენობის არგუმენტები შეგვიძლია გადავცეთ, აგრეთვე კონსოლური პროგრამის main() ფუნქციას. მოცემულ პროგრამაში:

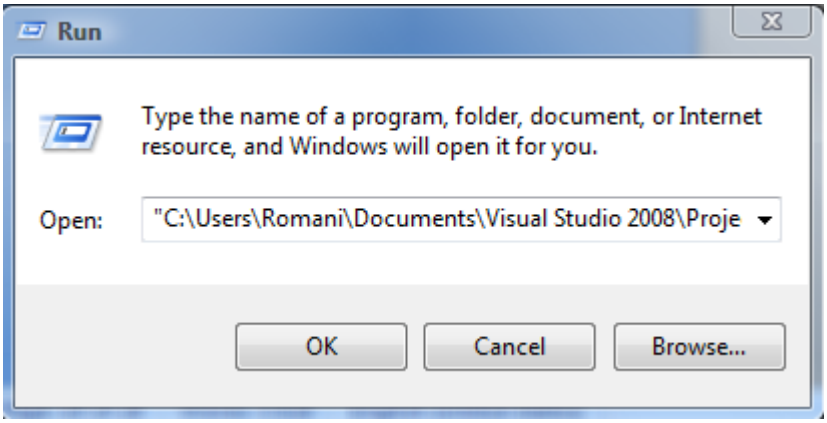
```
#include "stdafx.h"

using namespace System;
```

```
using namespace System::Threading;

int main(array<System::String ^> ^args)
{
Console::ReadLine();
Console::WriteLine(L" The number of arguments = {0}", args->Length);
int ricxvi = 1;
for each ( String^ striqoni in args )
        Console::WriteLine(L"argument {0} : {1}", ricxvi++, striqoni);
Thread::Sleep(3000);
return 0;
}
```

main() ფუნქციის args არგუმენტს ენიჭება მნიშვნელობები, რომლებიც შემდეგ ეკრანზე გამოგვაქვს. კომპილირების შემდეგ, ეს პროგრამა უნდა გაიშვას საბრძანებო სტრიქონიდან. მაგალითად, Start->Run (ნახ. 6.1). Browse კლავიშის საშუალებით ვირჩევთ საჭირო კატალოგს და .exe გაფართოების მქონე ფაილს. Open ველში გამოჩნდება არჩეული ფაილი, რომლის შემდეგ უნდა შევიტანოთ ინტერვალი, პარამეტრები და დავაჭიროთ OK კლავიშს. გახსნილ ფანჯარაში ვაჭერთ Enter კლავიშს და გამოჩნდება პროგრამის შესრულების შედეგები.



ნახ. 6.1.

ფუნქციის გადატვირთვა. პოლიმორფიზმი

C++ ენაში ერთი კლასის შიგნით ორ ან მეტ ფუნქციას შეიძლება ერთნაირი სახელი ჰქონდეს იმ პირობით, რომ ან პარამეტრების რაოდენობა უნდა იყოს განსხვავებული ან თუ პარამეტრების რაოდენობა ერთნაირია, მაშინ მათ სხვადასხვა ტიპი უნდა ჰქონდეს. ასეთ ფუნქციებს ***გადატვირთვადი*** ეწოდებათ, ხოლო ერთნაირი სახელის მქონე ფუნქციების განსაზღვრის პროცესს - ***ფუნქციის გადატვირთვა***. ფუნქციების გადატვირთვა არის პოლიმორფიზმის რეალიზების ერთ-ერთი საშუალება. გადატვირთვადი ფუნქციის გამოძახებისას სრულდება ის ფუნქცია, რომლის პარამეტრებიც ემთხვევა არგუმენტებს. გადატვირთვადი ფუნქციები შეიძლება აბრუნებდნენ სხვადასხვა ტიპის მონაცემებს. მაგალითი:

```
//      ფუნქციის გადატვირთვის დემონსტრირება
ref class Gadatvirtva {
//      ფუნქციას პარამეტრები არ აქვს და აბრუნებს სტრიქონს
public : System::String^ Funqcia() {
return L"გამოცხადებული იქნა უპარამეტრებო ფუნქცია";
```

```
}
//	ფუნქციას double ტიპის ერთი პარამეტრი აქვს და აბრუნებს მის კვადრატს
double Funqcia(double par1) {
	return System::Math::Pow(par1, 2);
}
//	ფუნქციას int ტიპის ორი პარამეტრი აქვს და აბრუნებს მათ ჯამს
int Funqcia(int par1, int par2) {
return par1 + par2;
}
//	ფუნქციას double ტიპის ორი პარამეტრი აქვს და აბრუნებს მათ ჯამს
double Funqcia(double par1, double par2) {
return par1 + par2;
}
};
//
private: System::Void button11_Click(System::Object^  sender, System::EventArgs^  e) {
Gadatvirtva^ obieqti;
int ricxvi1;
double wiladi1, wiladi2;
System::String^ striqoni;

//	Funqcia ფუნქციის ყველა ვერსიის გამოძახება
striqoni = obieqti.Funqcia();
label1->Text = striqoni;
wiladi2 = obieqti->Funqcia(5.7);
label2->Text = L"გამოძახებული იქნა ერთპარამეტრიანი ფუნქცია. შედეგი = " + wiladi2.ToString();
ricxvi1 = obieqti->Funqcia(5, 10);
label3->Text = L"გამოძახებული იქნა ორპარამეტრიანი ფუნქცია. შედეგი = " + ricxvi1.ToString();
wiladi1 = obieqti->Funqcia(5.5, 10.10);
label4->Text = L" გამოძახებული იქნა ორპარამეტრიანი ფუნქცია. შედეგი = " + wiladi1.ToString();
}
```

პროგრამაში ოთხჯერ ხდება Funqcia() ფუნქციის გადატვირთვა. პირველ ვერსიას არ აქვს პარამეტრი. მეორე ვერსიას აქვს ერთ მთელი ტიპის პარამეტრი, მესამეს - ორი მთელი ტიპის პარამეტრი და მეოთხეს - ორი წილადი პარამეტრი.

თუ პარამეტრების რაოდენობა თანაბარია და არგუმენტებსა და პარამეტრებს განსხვავებული ტიპები აქვს, მაშინ სრულდება ტიპების ავტომატური გარდაქმნა. განვიხილოთ მაგალითი:

```
//	ტიპების ავტომატური გარდაქმნა ფუნქციის გადატვირთვისას
ref class Gadatvirtva {
public : int Funqcia( int par1 ) {
	return par1;
}
double Funqcia( double par1 ) {
	return par1;
}
};
```

```
//
private: System::Void button10_Click(System::Object^  sender, System::EventArgs^  e) {
Gadatvirtva^ obieqti = gcnew Gadatvirtva();
int    i1 = 15;
double d1 = 15.15;
Byte   b1 = 33;
short  s1 = 20;
float  f1 = 5.5F;
//         აქ სრულდება Funqcia(int par1) ფუნქცია
label1->Text = obieqti->Funqcia(i1).ToString();
//         აქ სრულდება Funqcia(double par1) ფუნქცია
label2->Text = obieqti->Funqcia(d1).ToString();
//         აქ სრულდება Funqcia(int par1) ფუნქცია
label3->Text = obieqti->Funqcia(b1).ToString();
//         აქ სრულდება Funqcia(int par1) ფუნქცია
label4->Text = obieqti->Funqcia(s1).ToString();
//         აქ სრულდება Funqcia(double par1) ფუნქცია
label5->Text = obieqti->Funqcia(f1).ToString();
}
```

პროგრამაში Funqcia() ფუნქციის ერთ ვერსიას აქვს int ტიპის პარამეტრი, მეორეს კი - double ტიპის. მათი გამოძახება ხდება შესაბამისად int, double, byte, short და float ტიპის მქონე არგუმენტებისათვის. ფუნქციისთვის byte და short ტიპის მნიშვნელობების გადაცემისას ისინი ავტომატურად გარდაიქმნება int ტიპად, ხოლო float ტიპის მნიშვნელობის გადაცემისას კი - double ტიპად.

კონსტრუქტორის გადატვირთვა

ფუნქციის მსგავსად შესაძლებელია კონსტრუქტორის გადატვირთვა, დესტრუქტორის გადატვირთვა კი - არ შეიძლება. კონსტრუქტორის გადატვირთვას შემდეგი უპირატესობები აქვს: მოქნილობის უზრუნველყოფა, ობიექტების მასივების ინიციალიზება და ასლების კონსტრუქტორების შექმნა.

კონსტრუქტორების გადატვირთვა საშუალებას გვაძლევს ავირჩიოთ ობიექტის ინიციალიზების საშუალება. მოცემული პროგრამით ხდება გადატვირთვად კონსტრუქტორთან მუშაობის დემონსტრირება.

```
//         გადატვირთვადი კონსტრუქტორის დემონსტრირება
ref class ChemiKlasi1 {
public : int x;
ChemiKlasi1() {
x = 0;
}
ChemiKlasi1(int par1) {
x = par1 * par1;
}
ChemiKlasi1(double par2) {
x = (int) par2 + 10;
}
```

```
ChemiKlasi1(int par3, int par4) {
x = par3 * par4;
}
};
//
private: System::Void button16_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);

ChemiKlasi1^ obieqti1 = gcnew ChemiKlasi1();
label1->Text = "" + obieqti1->x.ToString();
ChemiKlasi1^ obieqti2 = gcnew ChemiKlasi1(ricxvi);
label2->Text = "" + obieqti2->x.ToString();
ChemiKlasi1^ obieqti3 = gcnew ChemiKlasi1(28.89);
label3->Text = "" + obieqti3->x.ToString();
ChemiKlasi1^ obieqti4 = gcnew ChemiKlasi1(3, 5);
label4->Text = "" + obieqti4->x.ToString();
}
```

პროგრამაში მოცემულია ChemiKlasi() ფუნქციის ოთხი ვერსია. საჭირო მათგანის არჩევა ხდება არგუმენტების მიხედვით.

ფუნქციების გადატვირთვა იძლევა იმის საშუალებას, რომ ერთი ობიექტის ინიციალიზებისათვის კონსტრუქტორმა მეორე ობიექტი გამოიყენოს. მოცემული პროგრამით ხდება ამის დემონსტრირება.

```
//      ერთი ობიექტის ინიციალიზებისათვის კონსტრუქტორი მეორე ობიექტს იყენებს
ref class Shekreba {
public : int jami;
//      ობიექტის შესაქმნელად კონსტრუქტორი იყენებს int ტიპის პარამეტრს
Shekreba(int raodenoba) {
jami = 0;
for ( int indexi = 1; indexi <= raodenoba; indexi++ )
jami += indexi;
}
//      ობიექტის ინიციალიზაციისათვის კონსტრუქტორი იყენებს სხვა ობიექტს,
//      რომელიც მას გადაეცემა, როგორც პარამეტრი
Shekreba(int par, Shekreba^ obieqti) {
jami = obieqti->jami;
}
};
//
private: System::Void button20_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
Shekreba^ obieqti1 = gcnew Shekreba(ricxvi1);
Shekreba^ obieqti2 = gcnew Shekreba(ricxvi1, obieqti1);

label1->Text = "";
label2->Text = "";
label1->Text = L"რიცხვების ჯამი 0-დან " + ricxvi1.ToString() + L"-მდე =  " +
                obieqti1->jami.ToString();
```

```
label2->Text = L"ცვლადის მნიშვნელობა =  " + obieqti2->jami.ToString();
}
```

პროგრამაში პირველი კონსტრუქტორის არგუმენტია მთელი რიცხვი, მეორე კონსტრუქტორის კი - ობიექტი. პირველი კონსტრუქტორი ანგარიშობს ჯამს ნულიდან textBox1 კომპონენტში შეტანილ რიცხვამდე. მეორე კონსტრუქტორი jami ცვლადს ანიჭებს მითითებული ობიექტის შესაბამისი ცვლადის მნიშვნელობას და აღარ ხდება საჭირო ჯამის მეორეჯერ გამოთვლა.

როგორც აღვნიშნეთ, კონსტრუქტორების გადატვირთვა იძლევა ობიექტების მასივის ინიციალიზების საშუალებას. მოცემული პროგრამით ხდება ერთი მასივის ინიციალიზება, მეორის კი - არა.

```
//      ობიექტების მასივის ინიციალიზება
ref class Gadatvirtva_1 {
int ricxvi;
public :
        Gadatvirtva_1() { ricxvi = 0; }                 //      ინიციალიზება არ ხდება
        Gadatvirtva_1(int par) { ricxvi = par; }        //      ინიციალიზება
        int Gacema() { return ricxvi; }
};
//
private: System::Void button12_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = L"obj1 მასივის ელემენტები - ";
label2->Text = L"obj2 მასივის ელემენტები - ";
int indexi;
//      ობიექტების შექმნა
array<Gadatvirtva_1^>^ obj1 = gcnew array<Gadatvirtva_1^> (5);
array<Gadatvirtva_1^>^ obj2 = gcnew array<Gadatvirtva_1^> (5);
//      ობიექტების ინიციალიზება
for ( indexi = 0; indexi < 5; indexi++ ) {
        obj1[indexi] = gcnew Gadatvirtva_1();           //      ინიციალიზება არ ხდება
        obj2[indexi] = gcnew Gadatvirtva_1(indexi);     //      ინიციალიზება
}
for ( indexi = 0; indexi < 5; indexi++ ) {
        label1->Text += obj1[indexi]->Gacema().ToString() + "   ";
        label2->Text += obj2[indexi]->Gacema().ToString() + "   ";
}
}
```

კონსტრუქტორების გადატვირთვა დაგვჭირდება ობიექტების მასივისთვის დინამიკური მეხსიერების გამოყოფის შემთხვევაშიც. როგორც ვიცით, დინამიკური მასივის ინიციალიზება არ შეიძლება. ამიტომ, თუ კლასი შეიცავს მაინიციალიზებელ კონსტრუქტორს, აუცილებელია, აგრეთვე, მისი არამაინიციალიზებელი გადატვირთული ვერსიის გამოცხადებაც. მოცემულ პროგრამაში ხდება ამის დემონსტრირება.

```
//
ref class Gadatvirtva {
int ricxvi;
public :
        Gadatvirtva() { ricxvi = 0; }
        Gadatvirtva(int par1) { ricxvi = par1; }
```

```
        int Gacema() { return ricxvi; }
        void Minicheba(int par2) { ricxvi = par2; }
};
//
private: System::Void button14_Click(System::Object^  sender, System::EventArgs^  e) {
label2->Text = L"obj2 მასივის ელემენტები - ";
int indexi, cvladi;
cvladi = Convert::ToInt32(textBox1->Text);
//      ერთი ობიექტის ინიციალიზება
Gadatvirtva^ obj1= gcnew Gadatvirtva(cvladi );
label1->Text = obj1->Gacema().ToString();
//      აქ ინიციალიზება არ შეიძლება
array<Gadatvirtva^>^ obj2 = gcnew array<Gadatvirtva^>(5);
if ( !obj2 ) label1->Text = L"მეხსიერების გამოყოფის შეცდომა!";
for ( indexi = 0; indexi < obj2->Length; indexi++ ) {
        obj2[indexi] = obj1;
}
for ( indexi = 0; indexi < 5; indexi++ ) {
        label2->Text += obj2[indexi]->Gacema().ToString() + "   ";
}
}
```

ასლის კონსტრუქტორი

გადატვირთული კონსტრუქტორის ერთ-ერთი ფორმაა ***ასლის კონსტრუქტორი*** (copy constructor). ის შემდეგი პრობლემების გადასაწყვეტად გამოიყენება:

- როგორც ვიცით, როდესაც ობიექტი ფუნქციას გადაეცემა, იქმნება ამ ობიექტის თანრიგობრივი ანუ ზუსტი ასლი, და იმ პარამეტრს გადაეცემა, რომელიც ობიექტს იღებს. მაგრამ, რიგ შემთხვევებში ობიექტი ზუსტი ასლის შექმნა არასასურველია. მაგალითად, თუ ობიექტი შეიცავს მიმთითებელს მეხსიერების გამოყოფილ უბანზე, მაშინ ასლში მიმთითებელი მიმართავს მეხსიერების იმავე უბანს. შედეგად, თუ ასლი ცვლის მეხსიერების უბნის შემცველობას, მაშინ ეს ცვლილებები შეეხება საწყის ობიექტსაც. გარდა ამისა, როდესაც ფუნქციის შესრულება მთავრდება, ასლი იშლება, რაც იწვევს ამ ასლის დესტრუქტორის გამოძახებას. დესტრუქტორის გამოძახებამ შეიძლება მიგვიყვანოს არასასურველ შედეგებამდე, რომლებიც საწყის ობიექტზე იმოქმედებენ.
- მსგავს სიტუაციას ადგილი აქვს მაშინ, როდესაც ფუნქცია გასცემს ობიექტს. დასაბრუნებელი ობიექტის შესანახად ავტომატურად იქმნება დროებითი ობიექტი. როგორც კი, მნიშვნელობა უბრუნდება გამომძახებელ პროგრამას, დროებითი ობიექტი გამოდის ხილვადობის უბნიდან, რაც იწვევს დროებითი ობიექტის დესტრუქტორის გამოძახებას. ამ დროს კი - შეიძლება წაიშალოს გამომძახებელი პროგრამისათვის საჭირო ინფორმაცია.

ამ პრობლემების საფუძვლია ობიექტის ზუსტი ასლის შექმნა. გამოსავალია ასლების კონსტრუქტორის გამოყენება, რომელიც მთლიანად აკონტროლებს აღნიშნულ პროცესებს. უნდა გვახსოვდეს, რომ C++ ენაში ერთი ობიექტის მნიშვნელობა მეორე ობიექტს შეიძლება გადაეცეს ორი გზით. პირველია ***მინიჭება***, მეორე კი ***ინიციალიზება***, რომელსაც შეიძლება ადგილი ჰქონდეს სამ შემთხვევაში:

- როდესაც გამოცხადებისას ერთი ობიექტი გამოიყენება მეორე ობიექტის

ინიციალიზებისთვის.

- როდესაც ობიექტი ფუნქციას გადაეცემა როგორც პარამეტრი.
- როდესაც ფუნქციის მიერ დაბრუნებულ მნიშვნელობას დროებითი ობიექტი წარმოადგენს.

ასლების კონსტრუქტორი გამოიყენება მხოლოდ ინიციალიზებისთვის და არ გამოიყენება მინიჭებისთვის. ინიციალიზების დროს ავტომატურად იქმნება ასლების კონსტრუქტორი, რომელიც ობიექტის ასლს ქმნის. ასლების კონსტრუქტორის საშუალებით შეგვიძლია წინასწარ განვსაზღვროთ თუ როგორ მოახდენს ერთი ობიექტი მეორის ინიციალიზებას. ასლების კონსტრუქტორის შექმნის შემდეგ ის ყოველთვის იქნება გამოძახებული ერთი ობიექტის მიერ მეორის ინიციალიზებისათვის.

ასლების კონსტრუქტორის სინტაქსია:

კლასის_სახელი (const *კლასის_სახელი* &*ობიექტის_სახელი*) {

// *კონსტრუქტორის კოდი*

}

მარტივი კლასებისთვის გამოიყენება გაჩუმებითი ასლის კონსტრუქტორი. მოცემული პროგრამით ხდება მასთან მუშაობის დემონსტრირება.

```
// გაჩუმებითი ასლის კონსტრუქტორთან მუშაობის დემონსტრირება
ref class Klasi {
        int ricxvi1;
        int ricxvi2;
public :
        Klasi(int par1, int par2) {
                ricxvi1 = par1;
                ricxvi2 = par2;
System::Windows::Forms::MessageBox::Show(L"მუშაობს ჩვეულებრივი კონსტრუქტორი");
        }
        void Naxva() {
        System::Windows::Forms::MessageBox::Show(ricxvi1.ToString() + "   " + ricxvi2.ToString());
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj1 = gcnew Klasi (ricxvi1, ricxvi2);
Klasi^ obj2 = obj1;                 //        სრულდება გაჩუმებითი ასლის კონსტრუქტორი
obj1->Naxva();
obj2->Naxva();
}
```

აქ კონსტრუქტორი ერთხელ გამოიძახება obj1 ობიექტის შექმნისას. რაც შეეხება obj2 ობიექტს, მისთვის კომპილატორმა შექმნა გაჩუმებითი ასლის კონსტრუქტორი. ასლის კონსტრუქტორი ქმნის obj2 ობიექტს და ახდენს მის ინიციალიზებას Klasi კლასის obj1 ობიექტით. ამ დროს, სრულდება არსებული obj1 ობიექტის წევრების გადაწერა ახალ obj2 ობიექტში.

თუ

```
Klasi^ obj2 = obj1;
```

სტრიქონის ნაცვლად ჩავწერთ

```
Klasi obj1(ricxvi1, ricxvi2);
Klasi obj2(ricxvi3, ricxvi4);
obj2 = obj1;
```
სტრიქონებს, მაშინ ასლების კონსტრუქტორი არ გამოიძახება.

რთული კლასებისთვის ჩვენ უნდა შევქმნათ საკუთარი ასლის კონსტრუქტორები. მაგალითი:

```
//      საკუთარ ასლის კონსტრუქტორთან მუშაობის დემონსტრირება
ref class Klasi {
        int ricxvi1;
        int ricxvi2;
public :
        Klasi(int par1, int par2) {
                ricxvi1 = par1;
                ricxvi2 = par2;
System::Windows::Forms::MessageBox::Show(L"მუშაობს ჩვეულებრივი კონსტრუქტორი");
        }
        Klasi(const Klasi^ obj) {
                ricxvi1 = obj->ricxvi1;
                ricxvi2 = obj->ricxvi2;
System::Windows::Forms::MessageBox::Show(L"მუშაობს ასლის კონსტრუქტორი");
        }
        void Naxva() {
        System::Windows::Forms::MessageBox::Show(ricxvi1.ToString() + "   " + ricxvi2.ToString());
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj1 = gcnew Klasi(ricxvi1, ricxvi2);
Klasi^ obj2 = gcnew Klasi(obj1);        //      სრულდება ჩვენს მიერ შექმნილი ასლის
კონსტრუქტორი
obj1->Naxva();
obj2->Naxva();
}
```

უნდა გვახსოვდეს, რომ ყოველთვის, როდესაც ადგილი აქვს მეხსიერების დინამიკურად გამოყოფას მშობლიური C++ პროგრამებისთვის, მაშინ ყოველთვის უნდა გამოვიყენოთ ასლის კონსტრუქტორი.

მიმართვა და ფუნქცია

მიმართვა შეგვიძლია სამნაირად გამოვიყენოთ:

- მიმართვა შეგვიძლია ფუნქციას გადავცეთ.
- მიმართვა შეგვიძლია ფუნქციიდან დავაბრუნოთ.
- შეგვიძლია შევქმნათ დამოუკიდებელი მიმართვა.

მიმართვა ხშირად გამოიყენება როგორც პარამეტრი ფუნქციისთვის გადასაცემად. იმის

გასარკვევად, თუ როგორ მუშაობს მიმართვა და რით განსხვავდება ის მიმთითებლისაგან, ჯერ განვიხილოთ მშობლიური C++ პროგრამა, რომლითაც ხდება მიმთითებელთან მუშაობის დემონსტრირება.

```
//          მიმთითებელთან მუშაობის დემონსტრირება
class Klasi {
public :
        void Funqcia(int *par, System::Windows::Forms::TextBox^ tb1) {
*par = System::Convert::ToInt32(tb1->Text);
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi;
Klasi *obj = new Klasi;

obj->Funqcia(&cvladi, textBox1);
label1->Text = cvladi.ToString();
}
```

ამ პროგრამაში cvladi ცვლადის მისამართი და textBox1 კომპონენტი Funqcia() ფუნქციას გადაეცემა. cvladi ცვლადის მისამართი მიენიჭება par პარამეტრს და ამ მისამართით ჩაიწერება textBox1 კომპონენტში მოთავსებული რიცხვი. პროგრამაში ნაჩვენებია, თუ როგორ უნდა გამოვიყენოთ მიმთითებელი მიმართვის მიხედვით (call by reference) ფუნქციისთვის არგუმენტის (ცვლადის) გადაცემის მექანიზმის რეალიზებისათვის. სწორედ, ამ მექნიზმის გამოყენებით შეგვიძლია შევასრულოთ ფუნქციის გამოძახება მიმართვის მიხედვით.

წინა პროგრამა შევცვალოთ ისე, რომ ფუნქციის პარამეტრად მივუთითოთ მიმართვა.

```
//          ფუნქციას პარამეტრად მიმართვა გადაეცემა
class Klasi {
public :
        void Funqcia(int &par, System::Windows::Forms::TextBox^ tb1)        {
                par = System::Convert::ToInt32(tb1->Text);
        }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi;
Klasi *obj = new Klasi;

obj->Funqcia(cvladi, textBox1);
label1->Text = cvladi.ToString();
}
```

როგორც პროგრამიდან ჩანს, მიმართვის გამოცხადებისათვის ვიყენებთ ამპერსანდს (&). ამიტომ, par პარამეტრი არის მიმართვა და მასთან მუშაობისას * ოპერატორს არ ვიყენებთ. par სახელი ავტომატურად განიხილება როგორც არგუმენტზე მიმთითებელი. ამიტომ,

```
par = System::Convert::ToInt32(tb1->Text);
```

მინიჭების შედეგად, textBox1 კომპონენტში შეტანილი მნიშვნელობა cvladi ცვლადს მიენიჭება. Funqcia() ფუნქციის გამოძახებისას მისი არგუმენტის წინ აღარ არის საჭირო & სიმბოლოს მითითება. რადგან, Funqcia() ფუნქცია გაცხადებულია, როგორც მიმართვის მიმღები, ამიტომ

მას *ავტომატურად* გადაეცემა არგუმენტის მისამართი.

უნდა გვახსოვდეს, რომ მისამართს, რომელზეც მიმართვა მიუთითებს, ჩვენ ვერ შევცვლით. მაგალითად, თუ Funqcia() ფუნქციას დავუმატებთ par--; ოპერატორს, მაშინ შესრულდება cvladi ცვლადის მნიშნელობის 1-ით შემცირება, და არა მისი მისამართის.

იგივე პროგრამის C++/CLI ვერსია:

```
ref class Klasi {
public :
        void Funqcia(int &par, System::Windows::Forms::TextBox^ tb1)       {
                par = System::Convert::ToInt32(tb1->Text);
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi;
Klasi^ obj = gcnew Klasi();

obj->Funqcia(cvladi, textBox1);
label1->Text = cvladi.ToString();
}
```

პარამეტრი-მიმართვების გამოყენებას პარამეტრი-მიმთითებლებთან შედარებით შემდეგი უპირატესობები აქვს:

- აღარ არის საჭირო მივიღოთ და ფუნქციას გადავცეთ არგუმენტის მისამართი. პარამეტრი-მიმართვების გამოყენებისას მისამართი ავტომატურად გადაიცემა.
- პარამეტრი-მიმართვები გვთავაზობენ უფრო გასაგებ და მარტივ ინტერფეისს, ვიდრე პარამეტრი-მიმთითებლები.
- ფუნქციისთვის ობიექტის გადაცემისას მიმართვის მიხედვით, ობიექტის ასლი არ იქმნება.

მოცემული მშობლიური C++ პროგრამით ხდება ორი მთელი რიცხვის მნიშვნელობების გაცვლა მიმართვის გამოყენებით.

```
//      ორი მთელი რიცხვის მნიშვნელობების      გაცვლა მიმართვის გამოყენებით
class Klasi {
public :
        void Funqcia(int &par1, int &par2)     {
                int temp;
                temp = par1;
                par1 = par2;
                par2 = temp;
        }
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi *obj = new Klasi;
label1->Text = cvladi1.ToString() + "   " + cvladi2.ToString();
obj->Funqcia(cvladi1, cvladi2);
label2->Text = cvladi1.ToString() + "   " + cvladi2.ToString();
}
```

```
	იმავე პროგრამის C++/CLI ვერსია:
ref class Klasi {
public :
	void Funqcia(int &par1, int &par2)	{
		int temp;
		temp = par1;
		par1 = par2;
		par2 = temp;
	}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj = gcnew Klasi();
label1->Text = cvladi1.ToString() + "   " + cvladi2.ToString();
obj->Funqcia(cvladi1, cvladi2);
label2->Text = cvladi1.ToString() + "   " + cvladi2.ToString();
}
```

ფუნქციისთვის ობიექტზე მიმართვების გადაცემა

ფუნქციისთვის მიმართვის მიხედვით ობიექტის გადაცემისას ობიექტის ასლი არ იქმნება და არც დესტრუქტორი გამოიძახება შედეგის გაცემისას. ამ შემთხვევაში, ფუნქციის შიგნით ობიექტის ცვლილების შემთხვევაში, იცვლება საწყისი ობიექტიც, რომელიც არგუმენტის სახით არის მითითებული.

უნდა გვახსოვდეს, რომ მიმართვა არ არის მიმთითებელი, თუმცა ის მიუთითებს ობიექტის მისამართზე. ამიტომ, მიმართვის მიხედვით ობიექტის გადაცემისას, მის წევრებთან მიმართვისათვის გამოიყენება ოპერატორი წერტილი (.) ოპერატორი ისრის (->) ნაცვლად.

მოცემული მშობლიური C++ პროგრამით ობიექტის გადაცემა ფუნქციისთვის ხდება პარამეტრი-მიმართვის გამოყენებით.

```
//	ფუნქციისთვის ობიექტის გადაცემა პარამეტრი-მიმართვის გამოყენებით
class Klasi {
	int ricxvi;
public :
Klasi(int par1) {
ricxvi = par1;
System::Windows::Forms::MessageBox::Show(L"მუშობს კონსტრუქტორი - " + ricxvi.ToString());
	}
~Klasi() {
System::Windows::Forms::MessageBox::Show(L"მუშობს დესტრუქტორი - " + ricxvi.ToString());
	}
	int Gamotana() {
		return ricxvi;
	}
};
void Funqcia(Klasi &obieqti) {
```

```
System::Windows::Forms::MessageBox::Show(L"მუშობს Funqcia - " + obieqti.Gamotana().ToString());
}
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
Klasi obj(cvladi);
Funqcia(obj);
}
```

იგივე პროგრამის C++/CLI ვერსია:

```
ref class Klasi {
	int ricxvi;
public :
Klasi(int par1) {
ricxvi = par1;
System::Windows::Forms::MessageBox::Show(L"მუშობს კონსტრუქტორი - " + ricxvi.ToString());
	}
~Klasi() {
System::Windows::Forms::MessageBox::Show(L"მუშობს დესტრუქტორი - " + ricxvi.ToString());
	}
	int Gamotana() {
		return ricxvi;
	}
};
void Funqcia(Klasi^ obieqti) {
System::Windows::Forms::MessageBox::Show(L"მუშობს Funqcia - " + obieqti->Gamotana().ToString());
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
Klasi^ obj = gcnew Klasi(cvladi);
Funqcia(obj);
delete obj;
}
```

ფუნქციიდან ობიექტზე მიმართვის დაბრუნება

ფუნქციას შეუძლია გასცეს მიმართვა. მოცემული მშობლიური C++ პროგრამით ხდება ფუნქციის მიერ მიმართვის დაბრუნების დემონსტრირება.

```
//	ფუნქციის მიერ მიმართვის დაბრუნების დემონსტრირება
class Klasi {
public :
	int ricxvi;
	int &Funqcia() {		//	ფუნქცია გასცემს მთელ რიცხვზე მიმართვას
		return ricxvi;		//	გაიცემა ricxvi-ზე მიმართვა
	}
};
//
```

```
private: System::Void button5_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
Klasi obj;

obj.Funqcia() = cvladi;
label1->Text = obj.ricxvi.ToString();
}
```

პროგრამაში Funqcia() ფუნქცია გასცემს მიმართვას მთელ რიცხვზე. ფუნქციაში return ricxvi; ოპერატორი არ გასცემს ricxvi ცვლადის მნიშვნელობას. ის ავტომატურად გასცემს ricxvi ცვლადის მისამართს მიმართვის სახით. აქედან გამომდინარე,
obj1.Funqcia() = cvladi;
ინსტრუქცია ricxvi ცვლადს ანიჭებს cvladi ცვლადის მნიშვნელობას.

ყურადღება მივაქციოთ იმას, რომ Funqcia() ფუნქცია მითითებულია მინიჭების ოპერატორის მარცხნივ. ამიტომ, მარცხნივ აღმოჩნდება მიმართვა იმ ობიექტზე, რომელსაც ეს ფუნქცია გასცემს. რადგან Funqcia() ფუნქცია გასცემს ricxvi ცვლადზე მიმართვას, ამიტომ ამ ცვლადს მიენიჭება cvladi ცვლადის მნიშვნელობა.

შეზღუდვები მიმართვების გამოყენებაზე

მიმართვების გამოყენებისას უნდა დავიცვათ შემდეგი შეზღუდვები:

- არ შეიძლება მივმართოთ სხვა მიმართვას.
- არ შეიძლება მივიღოთ მიმართვის მისამართი.
- არ შეიძლება შევქმნათ მიმართვების მასივი და მივმართოთ ბიტურ ველს.
- მიმართვა უნდა იყოს ინიციალიზებული მანამ, სანამ გახდება კლასის წევრი, დააბრუნებს ფუნქციის მნიშვნელობა ან გახდება ფუნქციის პარამეტრი.

ფუნქციის შაბლონი

ფუნქციის შაბლონი (function template) შეიცავს კოდს, რომელიც გამოყენებული იქნება სხვადასხვა ტიპის მონაცემების მიმართ. ფუნქციის შაბლონს სხვანაირად ***ფუნქცია-შაბლონს*** უწოდებენ. ის ოპერირებს იმ ტიპის მონაცემებზე, რომელიც მას გადაეცემა პარამეტრის სახით. ფუნქციის შაბლონის გამოყენების აუცილებლობა გამოწვეულია იმით, რომ ბევრი ალგორითმი ლოგიკურად ერთნაირია და შეიძლება შესრულდეს სხვადასხვა ტიპის მონაცემებზე. მაგალითად, მასივში მაქსიმალური ელემენტის პოვნის ალგორითმი ერთნაირია როგორც მთელრიცხვა მასივებისთვის, ისე წილადი რიცხვების მასივებისთვის. ფუნქციების შაბლონების გამოყენება საშუალებას გვაძლევს მონაცემების ტიპისგან დამოუკიდებლად განვსაზღვროთ ალგორითმი. ამის შემდეგ, კომპილატორი თვითონ აფორმირებს სწორ კოდს პარამეტრის ტიპის მიხედვით. ფაქტობრივად, ფუნქციის შაბლონი არის ფუნქცია, რომელიც ავტომატურად ასრულებს თვითგადატვირთვას.

ფუნქციის შაბლონის გამოცხადების სინტაქსია:

template < **class** ***ტიპი*** > ***დასაბრუნებელი_მნიშვნელობა ფუნქციის_სახელი*** (***პარამეტრების_სია***)
{
ფუნქციის კოდი
}

template (**შაბლონი**) სიტყვა მიუთითებს, რომ უნდა შეიქმნას ***ფუნქცია-შაბლონი***. ***ტიპი*** არის მონაცემების ტიპის ფიქტიური სახელი, რომელიც შეიცვლება მონაცემის რეალური ტიპის

სახელით ფუნქციის კონკრეტული ვერსიის შექმნისას.

როდესაც კომპილატორი ქმნის ფუნქციის შაბლონის კონკრეტულ ვერსიას, მაშინ ის ქმნის ***წარმოქმნილ ფუნქციას*** (generated function). წარმოქმნილი ფუნქციის გენერირების პროცესს ფუნქციის ***ეგზემპლარის შექმნა*** (instantiating) ეწოდება. ანუ წარმოქმნილი ფუნქცია არის ფუნქცია-შაბლონის კონკრეტული ეგზემპლარი. მოცემული პროგრამით ხდება ფუნქციის შაბლონთან მუშაობის დემონსტრირება

```
//      ფუნქციის შაბლონთან მუშაობის დემონსტრირება
//      ფუნქცია-შაბლონის გამოცხადება
template < class Tipi > Tipi Funqcia(Tipi par1, Tipi par2) {
        Tipi jami;

        jami = par1 + par2;
        return jami;
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
        int mteli1 = Convert::ToInt32(textBox1->Text);
        int mteli2 = Convert::ToInt32(textBox2->Text);
        double wiladi1 = Convert::ToDouble(textBox3->Text);
        double wiladi2 = Convert::ToDouble(textBox4->Text);

        int shedegi1 = Funqcia(mteli1, mteli2);
        double shedegi2 = Funqcia(wiladi1, wiladi2);
        label1->Text = shedegi1.ToString();
        label2->Text = shedegi2.ToString();
}
```

ამ პროგრამაში Tipi არის ზოგადი ტიპი, რომელიც გამოიყენება როგორც მონაცემების ტიპის ფიქტიური სახელი. როდესაც ხდება Funqcia(mteli1, mteli2) ფუნქციის გამოძახება, მაშინ Tipi იღებს int მნიშვნელობას, ხოლო Funqcia(wiladi1, wiladi2) ფუნქციის გამოძახებისას კი - double მნიშვნელობას.

class სიტყვის ნაცვლად შეგვიძლია გამოვიყენოთ typename სიტყვა:

```
template < typename Tipi > Tipi Funqcia_1(Tipi par1, Tipi par2) {
        Tipi jami;

        jami = par1 + par2;
        return jami;
}
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
        int mteli1 = Convert::ToInt32(textBox1->Text);
        int mteli2 = Convert::ToInt32(textBox2->Text);
        double wiladi1 = Convert::ToDouble(textBox3->Text);
        double wiladi2 = Convert::ToDouble(textBox4->Text);

        int shedegi1 = Funqcia_1(mteli1, mteli2);
        double shedegi2 = Funqcia_1(wiladi1, wiladi2);
        label1->Text = shedegi1.ToString();
```

```
	label2->Text = shedegi2.ToString();
}
```

ფუნქცია-შაბლონის განსაზღვრისას შეგვიძლია მივუთითოთ ერთ ზოგად ტიპზე მეტი. მაგალითი:

```
//	ფუნქციის შაბლონში მითითებულია ორი ზოგადი ტიპი
template < class Tipi_1, class Tipi_2 > Tipi_1 Funqcia_2(Tipi_1 par1, Tipi_2 par2) {
	Tipi_1 jami;

	jami = par1 + par2;
	return jami;
}
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
	int mteli1 = Convert::ToInt32(textBox1->Text);
	double wiladi1 = Convert::ToDouble(textBox3->Text);

	double shedegi1 = Funqcia_2(wiladi1, mteli1);
	label1->Text = shedegi1.ToString();
}
```

Funqcia_2(wiladi1, mteli1) ფუნქციის გამოძახებისას Tipi_1 იღებს double მნიშვნელობას, Tipi_2 კი - int მნიშვნელობას.

განსხვავება ფუნქცია-შაბლონებსა და გადატვირთულ ფუნქციებს შორის იმაში მდგომარეობს, რომ ფუნქცია-შაბლონები ერთსა და იმავე მოქმედებებს ასრულებენ სხვადასხვა ტიპის მონაცემებზე, ხოლო ფუნქციების გადატვირთვისას მათ შეიძლება განსხვავებული მოქმედებები შეასრულონ.

მართალია, ფუნქცია-შაბლონი თვითონ გადაიტვირთება, შესაძლებელია მისი აშკარად გადატვირთვაც. ამ შემთხვევაში, გადატვირთული ფუნქცია მალავს ფუნქცია-შაბლონს, თუ მოხდა პარამეტრების ტიპების დამთხვევა. მაგალითი:

```
template < class Tipi > Tipi Funqcia_3(Tipi par1, Tipi par2) {
	Tipi jami;

	jami = par1 + par2;
	return jami;
}
//	ფუნქცია-შაბლონის გადატვირთვა
double Funqcia_3(double par1, double par2) {
	double namravli;

	namravli = par1 * par2;
	return namravli;
}
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
	int mteli1 = Convert::ToInt32(textBox1->Text);
	int mteli2 = Convert::ToInt32(textBox2->Text);
	double wiladi1 = Convert::ToDouble(textBox3->Text);
	double wiladi2 = Convert::ToDouble(textBox4->Text);
```

```
        int shedegi1 = Funqcia_3(mteli1, mteli2);
        double shedegi2 = Funqcia_3(wiladi1, wiladi2);
        label1->Text = shedegi1.ToString();
        label2->Text = shedegi2.ToString();
}
```

Funqcia_3(wiladi1, wiladi2) ფუნქციის შესრულებისას გამოიძახება Funqcia_3(double par1, double par2) ფუნქცია. Funqcia_3(mteli1, mteli2) ფუნქციის შესრულებისას გამოიძახება Funqcia_3(Tipi par1, Tipi par2) ფუნქცია და Tipi ზოგადი ტიპი შეიცვლება int ტიპით.

კლასის შაბლონი

ფუნქციის შაბლონის გარდა, შესაძლებელია ***კლასის შაბლონის*** (***კლასი-შაბლონი***) განსაზღვრაც. კლასი-შაბლონში განსაზღვრულია ყველა ალგორითმი, ხოლო მონაცემების ფაქტიური ტიპები მოიცემა პარამეტრების სახით ამ კლასის ობიექტების შექმნისას.

კლასის შაბლონი სასარგებლოა მაშინ, როდესაც კლასი შეიცავს მუშაობის საერთო ლოგიკას. მისი საშუალებით შესაძლებელია ისეთი კლასის შექმნა, რომელიც ახდენს სტეკის, რიგის და ა.შ. ორგანიზებას.

კლასი-შაბლონის სინტაქსია:

template < class *ტიპი* > class *კლასის_სახელი* {

კლასის კოდი

}

კლასის შაბლონის შექმნის შემდეგ შეგვიძლია შევქმნათ ამ კლასის კონკრეტული ექზემპლარი შემდეგნაირად:

კლასის_სახელი < ტიპი > ობიექტი;

მოცემული პროგრამით ხდება კლასის შაბლონის შექმნის დემონსტრირება, რომელიც ახდენს სტეკის მუშაობის რეალიზებას მთელი რიცხვებისა და სიმბოლოებისათვის.

```
//      კლასის შაბლონის შექმნის დემონსტრირება
# define SIZE 10
//      კლასი-შაბლონი
template < class Steki_Tipi > class Steki
{
Steki_Tipi steki_masivi[SIZE];          //      სტეკი
int ind;                                //      სტეკის წვეროს ინდექსი
public :
        void init() { ind = 0; }        //      სტეკის ინიციალიზება
        void push(Steki_Tipi ch);
        Steki_Tipi pop();
};
//
template < class Steki_Tipi >
void Steki < Steki_Tipi >::push(Steki_Tipi ch)
{
if ( ind == SIZE )
{
System::Windows::Forms::MessageBox::Show(L"სტეკი სავსეა");
```

```
return;
}
steki_masivi[ind] = ch;
ind++;
}
//
template < class Steki_Tipi >
Steki_Tipi Steki < Steki_Tipi >::pop()
{
if ( ind == 0 )
{
System::Windows::Forms::MessageBox::Show(L"სტეკი ცარიელია");
return 0;
}
ind--;
return steki_masivi[ind];
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
//        სიმბოლოების და წილადების სტეკის შექმნა
Steki <Char> steki_1;
Steki <double> steki_2;
Char c1 = L'ა', c2 = L'ბ', c3 = L'ა', c4 = L'ს';
double d1 = 10.5, d2 = 9.03, d3 = 5.17;
//        სიმბოლოების სტეკის შექმნა
steki_1.init();
steki_1.push(c1);
steki_1.push(c2);
steki_1.push(c3);
steki_1.push(c4);
label1->Text += steki_1.pop().ToString();
label1->Text += steki_1.pop().ToString();
label1->Text += steki_1.pop().ToString();
label1->Text += steki_1.pop().ToString();

//        წილადების სტეკის შექმნა
steki_2.init();
steki_2.push(d1);
steki_2.push(d2);
steki_2.push(d3);
label2->Text += steki_2.pop().ToString() + "   ";
label2->Text += steki_2.pop().ToString() + "   ";
label2->Text += steki_2.pop().ToString();
}
```

პროგრამაში steki_1 კლასის გამოცხადებისას Steki კლასი-შაბლონის კოდში ყველგან Steki_Tipi ზოგადი ტიპი შეიცვლება Char ტიპით. ანალოგიურად, steki_2 კლასის გამოცხადებისას Steki კლასი-შაბლონის კოდში ყველგან Steki_Tipi ზოგადი ტიპი შეიცვლება

double ტიპით.

კლასის შაბლონის გამოცხადებისას შეიძლება მივუთითოთ რამდენიმე ზოგადი ტიპი. მაგალითი:

```
//      კლასების შაბლონში მითითებულია ორი ზოგადი ტიპი
template < class Tipi1, class Tipi2 > class ChemiKlasi {
Tipi1 cvladi1;
Tipi2 cvladi2;
public :
        ChemiKlasi(Tipi1 par1, Tipi2 par2) {
                cvladi1 = par1;
                cvladi2 = par2;
        }
        void Naxva() {
        System::Windows::Forms::MessageBox::Show(L"პირველი ცვლადის მნიშვნელობაა = " +
        cvladi1.ToString() + L"\nმეორე ცვლადის მნიშვნელობაა = " + cvladi2.ToString());
        }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
        int mteli = Convert::ToInt32(textBox1->Text);
        double wiladi = Convert::ToDouble(textBox2->Text);
        ChemiKlasi <double, int> obj1(wiladi, mteli);
        obj1.Naxva();
}
```

განზოგადებული ფუნქცია

განზოგადებული ფუნქცია (generic, обобщенный) ფუნქციის შაბლონის მსგავსია, მაგრამ განსახვავებულად მუშაობს. როდესაც ვიყენებთ ფუნქციის შაბლონს, მაშინ კომპილატორი ახდენს ფუნქციების კონკრეტული ეგზემპლარების საწყისი კოდის გენერირებას. შემდეგ, სრულდება ამ გენერირებული საწყისი კოდის კომპილირება პროგრამის დანარჩენ კოდთან ერთად. რიგ შემთხვევებში, ეს იწვევს დიდი რაოდენობის ფუნქციების გენერირებას და შესაბამისად ზრდის შესრულებადი ფაილის ზომას. განზოგადებული ფუნქციის გამოყენების შემთხვევაში, მისი კოდი ჩვეულებრივ კომპილირდება, და როდესაც მოხდება ფუნქციის გამოძახება, მაშინ შესრულდება ტიპის პარამეტრების ჩასმა შესრულებისას. შესრულებისას არ გენერირდება დამატებითი კოდი და შესაბამისად, შესრულებადი კოდის ზომა არ იზრდება.

განზოგადებული ფუნქციის გამოცხადების სინტაქსია:

generic<typename *ტიპის_სახელი*> **where** *პარამეტრი_ტიპი* **:** *შეზღუდვების_სია*
ტიპის_სახელი ფუნქციის_სახელი (**array<***ტიპის_სახელი*>^ *პარამეტრი*)
{
ფუნქციის კოდი
}

მოცემული პროგრამით ხდება განზოგადებულ ფუნქციასთან მუშაობის დემონსტრირება.

```
//      განზოგადებულ ფუნქციასთან მუშაობის დემონსტრირება
generic<typename Tipi> where Tipi : System::IComparable
Tipi Funqcia(array<Tipi>^ masivi)
```

```
{
Tipi max = masivi[0];
for ( int indexi = 1; indexi < masivi->Length; indexi++ )
        if ( max->CompareTo(masivi[indexi]) < 0 ) max = masivi[indexi];
return max;
}
//
generic<typename Tipi_2> where Tipi_2 : System::IComparable
void Funqcia2(array<Tipi_2>^ masivi2, System::Windows::Forms::Label^ lab2)
{
        for each ( Tipi_2 x in masivi2 )
                lab2->Text += x + "   ";
        lab2->Text += "\n";
}
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
array<int>^ masivi1 = { 1, 2, 3, 4, 5 };
array<double>^ masivi2 = { 1.1, 2.2, 3.3, 4.4 };
array<String^>^ masivi3 = { L"საბა", L"ანა", L"ლია" };

Funqcia2(masivi1, label1);
Funqcia2(masivi2, label2);
Funqcia2(masivi3, label3);
label1->Text += Funqcia(masivi1);
label2->Text += Funqcia(masivi2);
label3->Text += Funqcia(masivi3);
}
```

აქ IComparable არის ინტერფეისი, რომელსაც მე-8 თავში განვიხილავთ. Tipi არის განზოგადებული ფუნქციის ტიპის პარამეტრის სახელი. ტიპის პარამეტრი შეიცვლება ნამდვილი ტიპით მოცემული ფუნქციის გამოძახებისას. თუ მითითებულია რამდენიმე ტიპის პარამეტრი, მაშინ ისინი ერთმანეთისგან მძიმეებით უნდა გამოიყოს.

where საკვანძო სიტყვა განსაზღვრავს შეზღუდვებს, რომლებიც ედება იმ ტიპს, რომელმაც უნდა შეცვალოს ტიპის სახელი განზოგადებული ფუნქციის გამოძახებისას. ეს შეზღუდვა იმას ნიშნავს, რომ ნებისმიერმა ტიპმა, რომელიც ტიპის პარამეტრს ცვლის, უნდა მოახდინოს IComparable ინტერფეისის რელიზება. კერძოდ, ჩასასმელმა ტიპმა უნდა განსაზღვროს CompareTo() ფუნქცია, რომელიც ადარებს მოცემული ტიპის ორ ობიექტს. ამ შეზღუდვის გარეშე კომპილატორს არ ეცოდინება თუ რა ოპერაციები უნდა შესრულდეს ნამდვილი ტიპის მიმართ. ეს ფუნქცია გასცემს 0-ს, თუ ობიექტები ტოლია, დადებით მნიშვნელობას, თუ პირველი ობიექტი მეორეზე მეტია და უარყოფით მნიშვნელობას თუ პირველი ობიექტი მეორეზე ნაკლებია.

სტრიქონი:

Tipi Funqcia(array<Tipi>^ masivi)

განსაზღვრავს Funqcia ფუნქციას, რომელის გასცემს Tipi ტიპის შედეგს.

ძირითადი პროგრამის

Funqcia2(masivi1, label1);

სტრიქონში ხდება Funqcia2() ფუნქციის გამოძახება, რომელსაც პარამეტრად მთელრიცხვა მასივი გადაეცემა. განზოგადებულ ფუნქციაში Tipi ტიპი შეიცვლება int ტიპით. ფუნქცია

იმუშავებს მთელრიცხვა მასივისთვის და შედეგად გასცემს მთელ რიცხვს. ანალოგიურად,
Funqcia2(masivi2, label2);
სტრიქონში Funqcia2() ფუნქციას გადაეცემა წილადების მასივი და Tipi ტიპი შეიცვლება double ტიპით. შესაბამისად, ფუნქცია იმუშავებს წილად რიცხვებთან და გასცემს წილად შედეგს.
Funqcia2(masivi3, label3);
სტრიქონში Funqcia2() ფუნქციას გადაეცემა სტრიქონების მასივი და Tipi ტიპი შეიცვლება String ტიპით. შესაბამისად, ფუნქცია იმუშავებს სტრიქონებთან და გასცემს სტრიქონს.

განზოგადებული კლასი

C++/CLI ენა იძლევა ***განზოგადებული კლასის*** განსაზღვრის შესაძლებლობას. განზოგადებული კლასის ტიპისგან პროგრამის შესრულებისას შეგვიძლია შევქმნათ სპეციფიკური კლასი. შეგვიძლია განვსაზღვროთ მნიშვნელობების განზოგადებული კლასები, განზოგადებული მიმართვითი კლასები, განზოგადებული ინტერფეისული კლასები და განზოგადებული დელეგატები (დელეგატებს მე-8 თავში განვიხილავთ).

განზოგადებული კლასის გამოცხადების სინტაქსია:

generic<typename *კლასის_ტიპი*> ref class *კლასის_სახელი*
{
კლასის კოდი
};
ან
generic<class *კლასის_ტიპი*> ref class *კლასის_სახელი*
{
კლასის კოდი
};

როგორც ვხედავთ **typename** საკვანძო სიტყვის ნაცვლად შეგვიძლია გამოვიყენოთ **class** საკვანძო სიტყვა.

განზოგადებული კლასის გამოცხადების მაგალითია:

```
generic<class Tipi> ref class Steki
{
//      კლასის კოდი
};
```

Steki განზოგადებული კლასის რეალიზების მაგალითია:

```
Steki<Klasi^>^ obj1 = gcnew Steki<Klasi^>;
Steki<int>^ obj2 = gcnew Steki<int>;
Steki<Char>^ obj3 = gcnew Steki<Char>;
```

obj1 ობიექტისთვის Klasi^ არგუმენტი-ტიპი ცვლის კლასის ტიპს კლასის განსაზღვრაში. obj2 ობიექტისთვის int არგუმენტი-ტიპი ცვლის კლასის ტიპს კლასის განსაზღვრაში. obj3 ობიექტისთვის Char არგუმენტი-ტიპი ცვლის კლასის ტიპს კლასის განსაზღვრაში.

მოცემული პროგრამით ხდება განზოგადებულ კლასთან მუშაობის დემონსტრირება.

```
//      განზოგადებულ კლასთან მუშაობის დემონსტრირება
generic<class Tipi> ref class Steki
{
private :
        //
        ref struct Item
```

```
        {
                Tipi Obj;
                Item^ Next;
                Item(Tipi obj, Item^ next) : Obj(obj), Next(next) { }
        };
        Item^ Top;
public :
        void Push(Tipi obj)
        {
                Top = gcnew Item(obj, Top);
        }
        Tipi Pop()
        {
                if ( Top == nullptr ) return Tipi();
                Tipi obj = Top->Obj;
                Top = Top->Next;
                return obj;
        }
};
//
ref class Klasi {
public :
        int ricxvi;
        Klasi(int par) { ricxvi = par; }
        int Naxva() { return ricxvi * 2; }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
 Steki<Klasi^>^ obj1 = gcnew Steki<Klasi^>;
 Steki<int>^ obj2 = gcnew Steki<int>;
 Steki<Char>^ obj3 = gcnew Steki<Char>;
 Klasi^ k1 = gcnew Klasi(1); Klasi^ k2 = gcnew Klasi(2); Klasi^ k3 = gcnew Klasi(3);
Char c1 = L'ა', c2 = L'ბ', c3 = L'ა', c4 = L'ს';
int i1 = 10, i2 = 5, i3 = 15;
//
obj1->Push(k1);
obj1->Push(k2);
obj1->Push(k3);
label1->Text = obj1->Pop()->ricxvi.ToString() + "  ";
label1->Text += obj1->Pop()->ricxvi.ToString() + "  ";
label1->Text += obj1->Pop()->ricxvi.ToString();
obj1->Pop();
//
obj2->Push(i1);
obj2->Push(i2);
obj2->Push(i3);
label2->Text = obj2->Pop().ToString() + "   ";
```

```
label2->Text += obj2->Pop().ToString() + "   ";
label2->Text += obj2->Pop().ToString();
//
obj3->Push(c1);
obj3->Push(c2);
obj3->Push(c3);
obj3->Push(c4);
label3->Text = obj3->Pop().ToString();
label3->Text += obj3->Pop().ToString();
label3->Text += obj3->Pop().ToString();
label3->Text += obj3->Pop().ToString();
}
```

ჩასადგმელი ფუნქცია

C++ ენაში შეგვიძლია გამოვაცხადოთ ფუნქცია, რომელიც არ გამოიძახება. პროგრამაში მისი კოდის მოთავსება ხდება გამოძახების ადგილში. ***ჩასადგმელი ფუნქციის*** (in-line) უპირატესობა იმაში მადგომარეობს, რომ ის დაკავშირებული არ არის ფუნქციების გამოძახებისა და მნიშვნელობის დაბრუნების მექანიზმთან. ამიტომ, ჩასადგმელი ფუნქციები სწრაფად სრულდება. ასეთი ფუნქციების ნაკლია ის, რომ თუ ისინი დიდი ზომისაა და ხშირად გვხვდება პროგრამაში, მაშინ პროგრამის ზომა მკვეთრად იზრდება. ამის გამო, ჩასადგმელი ფუნქციები ხშირად გამოიყენება, მაშინ როდესაც ფუნქციის ზომა მცირეა. ჩასადგმელი ფუნქციის გამოცხადებისათვის ფუნქციის სახელის წინ უნდა მივუთითოთ **inline** სპეციფიკატორი.

მოცემული პროგრამით ხდება ჩასადგმელ ფუნქციასთან მუშაობის დემონსტრირება.

```
//      inline ფუნქციასთან მუშაობა
ref class Klasi {
int ricxvi1, ricxvi2;
public :
        Klasi(int par1, int par2);
        int Jami();                             //      ჩადგმა სრულდება ამ გამოცხადებაში
};
Klasi::Klasi(int par1, int par2) {
        ricxvi1 = par1;
        ricxvi2 = par2;
}
inline int Klasi::Jami() {                      //      Jami() ფუნქცია არის ჩასადგმელი
        return ricxvi1 + ricxvi2;
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj =  gcnew Klasi(cvladi1, cvladi2);
int jami = obj->Jami();
label1->Text = jami.ToString();
}
```

ამ პროგრამაში Jami() ფუნქცია გამოცხადებულია როგორც ჩასადგმელი. ეს იმას ნიშნავს, რომ int jami = obj->Jami(); სტრიქონში ფუნქციის გამოძახება არ მოხდება, რადგან მინიჭების ოპერატორის მარჯვენა ნაწილში მოთავსებული იქნება Jami() ფუნქციის კოდი.

ჩასადგმელი ფუნქცია გამოცხადებული უნდა იყოს მის პირველ გამოყენებამდე. წინააღმდეგ შემთხვევაში, კომპილატორს არ ეცოდინება, თუ რა კოდი უნდა ჩასვას პროგრამაში.

inline სპეციფიკატორი კომპილატორისთვის წარმოადგენს ***მოთხოვნას*** და არა ბრძანებას. თუ კომპილატორმა ვერ შეძლო მოთხოვნის შესრულება, მაშინ **inline** მოთხოვნა უარიყოფა და ფუნქცია კომპილირდება, როგორც ჩვეულებრივი ფუნქცია. ზოგიერთი კომპილატორი ფუნქციას არ აღიქვამს როგორც ჩასადგმელს, თუ ფუნქცია შეიცავს რეკურსიას, ან სტატიკურ წევრს და ა.შ.

თუ ფუნქცია და მისი კოდი გამოცხადებულია კლასის შიგნით, მაშინ ეს ფუნქცია ავტომატურად ხდება ჩასადგმელი ფუნქცია და **inline** სპეციფიკატორის მითითება აუცილებელი არ არის. მაგალითი:

```
//
ref class Klasi {
int ricxvi1, ricxvi2;
public :
	Klasi(int par1, int par2)	{
		ricxvi1 = par1;
		ricxvi2 = par2;
	}
	int Jami() {					//	Jami() ფუნქცია არის ჩასადგმელი
		return ricxvi1 + ricxvi2;
	}
};
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj = gcnew Klasi(cvladi1, cvladi2);
int jami = obj->Jami();
label1->Text = jami.ToString();
}
```

ბოლოს აღვნიშნოთ, რომ ჩასადგმელი ფუნქციები შეიძლება იყოს გადატვირთული.მაგალითი:

```
//	ჩასადგმელი ფუნქციების გადატვირთვის დემონსტრირება
ref class Klasi {
int ricxvi1, ricxvi2;
public :
	int Jami(int par1, int par2)	{
		return par1 + par2;
	}
	double Jami(double par1, double par2)	{
		return par1 + par2;
	}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
Klasi^ obj = new Klasi();
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
double cvladi3 = Convert::ToDouble(textBox3->Text);
double cvladi4 = Convert::ToDouble(textBox4->Text);
int jami_int = obj->Jami(cvladi1, cvladi2);
double jami_double = obj->Jami(cvladi3, cvladi4);
label1->Text = jami_int.ToString();
label2->Text = jami_double.ToString();
}
```

მეგობრული ფუნქცია

მეგობრულია ფუნქცია (friend function), რომელიც არ არის კლასის წევრი, მაგრამ შეუძლია მიმართოს კლასის დახურულ წევრებს. მეგობრული მეოდების შემოტანა განპირობებულია იმით, რომ მათ შეუძლიათ მიმართონ რამდენიმე კლასის დახურულ წევრებს, ისინი გამოიყენება ოპერატორების გადატვირთვისას და შეტანა-გამოტანის სპეციალური ფუნქციების შექმნისას.

მეგობრული ფუნქცია განისაზღვრება ჩვეულებრივი ფუნქციის მსგავსად, ოღონდ მისი სახელის წინ უნდა მივუთითოთ **friend** სიტყვა და მეგობრული ფუნქციის პროტოტიპი უნდა ჩავრთოთ კლასის გამოცხადებაში. მოცემული მმობლიური C++ პროგრამით ხდება მეგობრულ ფუნქციასთან მუშაობის დემონსტრირება.

// მეგობრულ ფუნქციასთან მუშაობის დემონსტრირება

```
class Klasi_1 {
int ricxvi1, ricxvi2, jami;
public :
        Klasi_1(int par1, int par2) {
                ricxvi1 = par1;
                ricxvi2 = par2;
        }
        friend int Megobruli_Funqcia(Klasi_1 obj2);
};
int Megobruli_Funqcia(Klasi_1 obj2) {
        obj2.jami = obj2.ricxvi1 + obj2.ricxvi2;
        return obj2.jami;
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
int cvladi4 = Convert::ToInt32(textBox4->Text);
Klasi_1 obj1(cvladi1, cvladi2);
Klasi_1 obj2(cvladi3, cvladi4);
int shedegi1 =  Megobruli_Funqcia(obj1);
int shedegi2 =  Megobruli_Funqcia(obj2);
```

```
label1->Text = shedegi1.ToString();
label2->Text = shedegi2.ToString();
}
```

როგორც პროგრამიდან ჩანს, Megobruli_Funqcia მეგობრული ფუნქცია კლასში განსაზღვრული ფუნქციებისაგან განსხვავებით კლასის დახურულ წევრებს მიმართავს ამავე კლასის ობიექტის საშუალებით. საქმე ის არის, რომ მეგობრული ფუნქცია დაკავშირებული არ არის რაიმე ობიექტთან, რადგან ის არ არის კლასის წევრი. ამიტომ, ის ვერ იმუშავებს კონკრეტული ობიექტის დახურულ ცვლადთან ამ ობიექტის სახელის მითითების გარეშე.

მეგობრული ფუნქცია არ არის კლასის წევრი და არ შეიძლება გამოვიძახოთ ობიექტის სახელის საშუალებით. ის ისევე უნდა გამოვიძახოთ, როგორ ჩვეულებრივი ფუნქცია, ანუ როგორც კლასის გარეთ განსაზღვრული ფუნქცია.

ფუნქცია მეგობრული შეიძლება იყოს რამდენიმე კლასის მიმართ. მაგალითი:

```
//      წინმსწრები მიმართვის დემონსტრირება
class Injineri;             //      წინმსწრები მიმართვა
class Eqimi {
int staji;
double xelfasi;
public :
int ricxvi1;
Eqimi(int par1, double par2, int par3) {
staji = par1; xelfasi = par2; ricxvi1 = par3;
}
friend int MegobruliFunqcia(Eqimi obj1, Injineri obj2);
};
//
class Injineri {
int staji;
double xelfasi;
public :
int ricxvi2;
Injineri(int par1, double par2, int par3) {
staji = par1; xelfasi = par2; ricxvi2 = par3;
}
friend int MegobruliFunqcia(Eqimi obj1, Injineri obj2);
};
int MegobruliFunqcia(Eqimi obj1, Injineri obj2) {
        return obj1.staji - obj2.staji;
}
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToDouble(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
int cvladi4 = Convert::ToInt32(textBox4->Text);
int cvladi5 = Convert::ToDouble(textBox5->Text);
int cvladi6 = Convert::ToInt32(textBox6->Text);
Eqimi obj1(cvladi1, cvladi2, cvladi3);
```

```
Injineri obj2(cvladi4, cvladi5, cvladi6);
int shedegi = MegobruliFunqcia(obj1, obj2);
if ( shedegi > 0 ) label1->Text = L"ექიმს ინჟინერზე მეტი სტაჟი აქვს";
        else if ( shedegi < 0 ) label1->Text = L"ექიმს ინჟინერზე ნაკლები სტაჟი აქვს";
                                    else label1->Text = L"ექიმს და ინჟინერს ერთნაირი სტაჟი აქვს";
}
```

MegobruliFunqcia() ფუნქცია მეგობრულია Eqimi და Injineri კლასების მიმართ. ეს ფუნქცია ექიმის სტაჟს აკლებს ინჟინრის სტაჟს და გასცემს სხვაობას. ეს სხვაობა მოწმდება ძირითად პროგრამაში და გაიცემა შესაბამისი შეტყობინება.

პროგრამაში ხდება ***წინასწარი გამოცხადების*** (forward declaration) დემონსტრირება, რომელსაც სხვანაირად ***წინმსწრები მიმართვა*** (forward reference) ეწოდება. რადგან, MegobruliFunqcia() ფუნქცია ორივე კლასის პარამეტრს იღებს, ამიტომ ლოგიკურად შეუძლებელია ორივე კლასის გამოცხადება თითოეულ მათგანში ამ ფუნქციის ჩართვამდე. ამიტომ, კომპილატორს უნდა შევატყობინოთ კლასის სახელი მის გამოცხადებამდე. სწორედ ასეთ საშუალებას ეწოდება წინასწარი გამოცხადება. მისი სინტაქსია:

class *კლასის_სახელი;*

ჩვენს მაგალითში Eqimi კლასის გამოცხადების წინ მითითებულია Injineri კლასის სახელი. ამის შემდეგ მეგობრული ფუნქციის გამოცხადებაში შეგვიძლია გამოვიყენოთ Injineri კლასი.

ფუნქცია შეიძლება იყოს ერთი კლასის წევრი და მეგობრული მეორე კლასისთვის. მაგალითი:

```
//      MegobruliFunqcia ფუნქცია არის Eqimi კლასის წევრი
//      და მეგობრული Injineri კლასისთვის
class Injineri;         //      წინმსწრები გამოცხადება
class Eqimi
{
int staji;
double xelfasi;
public :
int ricxvi1;
Eqimi(int par1, double par2, int par3)
{
staji = par1; xelfasi = par2; ricxvi1 = par3;
}
int MegobruliFunqcia(Injineri obj2);
};
//
class Injineri
{
int staji;
double xelfasi;
public :
int ricxvi2;
Injineri(int par1, double par2, int par3)
{
staji = par1; xelfasi = par2; ricxvi2 = par3;
}
```

```
friend int Eqimi::MegobruliFunqcia(Injineri obj2);
};
int Eqimi::MegobruliFunqcia(Injineri obj2)
{
	return staji - obj2.staji;
}
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToDouble(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
int cvladi4 = Convert::ToInt32(textBox4->Text);
int cvladi5 = Convert::ToDouble(textBox5->Text);
int cvladi6 = Convert::ToInt32(textBox6->Text);
Eqimi obj1(cvladi1, cvladi2, cvladi3);
Injineri obj3(cvladi4, cvladi5, cvladi6);
int shedegi = obj1.MegobruliFunqcia(obj3);
if ( shedegi > 0 ) label1->Text = L"ექიმს აქვს ინჟინერზე მეტი სტაჟი";
	else if ( shedegi < 0 ) label1->Text = L"ექიმს აქვს ინჟინერზე ნაკლები სტაჟი";
				else label1->Text = L"ექიმს და ინჟინერს ერთნაირი სტაჟი აქვთ";
}
```

Injineri კლასში friend int Eqimi::MegobruliFunqcia(Injineri obj2); გამოცხადება მიუთითებს, რომ MegobruliFunqcia ფუნქცია არის Eqimi კლასის წევრი.

მეგობრული ფუნქცია მემკვიდრეობით არ გადაიცემა. თუ საბაზო კლასის გამოცხადებაში ჩართულია მეგობრული ფუნქციის პროტოტიპი, მაშინ ის მეგობრულია მხოლოდ ამ საბაზო კლასის მიმართ და არ არის მეგობრული მემკვიდრე კლასების მიმართ (მემკვიდრეობითობას მე-8 თავში განვიხილავთ).

ბოლოს, შევნიშნოთ რომ, არ შეიძლება მეგობრული ფუნქციების განსაზღვრა ref და value კლასებში.

ნაგულისხმევი არგუმენტი

ნაგულისხმევი არგუმენტი (default argument) გამოიყენება ფუნქციის გამოძახებისას პარამეტრისთვის წინასწარ განსაზღვრული მნიშვნელობის მისანიჭებლად იმ შემთხვევაში, როდესაც შესაბამისი არგუმენტი მითითებული არ არის. ნაგულისხმევი არგუმენტის გამოყენება არის ფუნქციის გადატვირთვის ფარული ფორმა.

პარამეტრს რომ გადავცეთ ნაგულისხმევი არგუმენტი, ამისათვის ფუნქციის გამოცხადებისას უნდა შევასრულოთ პარამეტრის ინიციალიზება, ანუ საჭირო პარამეტრს უნდა მივანიჭოთ საჭირო მნიშვნელობა. მაგალითად, გამოვაცხადოთ ორ პარამეტრიანი ფუნქცია და ორივე პარამეტრს მივანიჭოთ ჩვენთვის სასურველი მნიშვნელობები:

```
int Funqcia(int par1 = 5, int par2 = 20);
```

ეს ფუნქცია სამი გზით შეგვიძლია გამოვიძახოთ:

```
int shedegi1 = Funqcia();
int shedegi2 = Funqcia(7);
int shedegi3 = Funqcia(2, 8);
```

პირველ შემთხვევაში, არგუმენტები არა არის მითითებული და იგულისხმება ფუნქციის გამოცხადებისას განსაზღვრული მნიშვნელობები. მეორე შემთხვევაში, მითითებულია ერთი

არგუმენტი, რომელიც par1 პარამეტრს მიენიჭება, par2 პარამეტრს კი - მიენიჭება მნიშვნელობა 20. მესამე შემთხვევაში, par1 პარამეტრს მიენიჭება მნიშვნელობა 2, par2 პარამეტრს კი - მნიშვნელობა 8.

უნდა გვახსოვდეს, რომ თუ პირველი არგუმენტი მოიცემა გაჩუმებით, მაშინ ყველა დანარჩენი პარამეტრიც გაჩუმებით უნდა მოიცეს. მაგალითად, არასწორია ფუნქციის ასეთი გამოცხადება:

```
int Funqcia(int par1 = 5, int par2);
```

სწორია ფუნქციის შემდეგი გამოცხადება:

```
int Funqcia(int par1, int par2 = 9);
```

მოცემული მშობლიური C++ პროგრამით ხდება გაჩუმებით განსაზღვრულ არგუმენტებთან მუშობის დემონსტრირება.

```
//      გაჩუმებით განსაზღვრულ არგუმენტებთან მუშობის დემონსტრირება
class Klasi {
public :
        int Funqcia(int par1 = 5, int par = 10);
};
int Klasi::Funqcia(int par1, int par2) {
        return par1 + par2;
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi obj1;
int shedegi1 = obj1.Funqcia();
int shedegi2 = obj1.Funqcia(cvladi1);
int shedegi3 = obj1.Funqcia(cvladi1, cvladi2);
label1->Text = shedegi1.ToString();
label2->Text = shedegi2.ToString();
label3->Text = shedegi3.ToString();
}
```

ნაგულისხმევი არგუმენტები დაკავშირებულია ფუნქციების გადატვირთვასთან. ეს ნაჩვენებია მოცემულ მშობლიურ C++ პროგრამაში. კლასში ორი ფუნქციაა განსაზღვრული. პირველი მათგანი გამოთვლის მართკუთხედის პერიმეტრს, მეორე კი - სამკუთხედის პერიმეტრს.

```
//      ნაგულისხმევი არგუმენტებთან მუშაობის დემონსტრირება
class Klasi1 {
public :
        int Perimetri(int par1, int par2, int par3) {
                return par1 + par2 + par3;
        }
        int Perimetri(int par1, int par2) {
                return par1 + par2;
        }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
Klasi1 obj1;
int perimetri_samkutxedi = obj1.Perimetri(cvladi1, cvladi2, cvladi3);
int perimetri_martkutxedi = obj1.Perimetri(cvladi1, cvladi2);
label1->Text = perimetri_samkutxedi.ToString();
label2->Text = perimetri_martkutxedi.ToString();
}
```

ეს პროგრამა ნაგულისხმევი არგუმენტების გამოყენებით შეგვიძლია ასე ჩავწეროთ.

```
class Klasi2 {
public :
        int Perimetri(int par1, int par2, int par3 = 0) {
                return par1 + par2 + par3;
        }
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
Klasi2 obj1;
int perimetri_samkutxedi = obj1.Perimetri(cvladi1, cvladi2, cvladi3);
int perimetri_martkutxedi = obj1.Perimetri(cvladi1, cvladi2);
label1->Text = perimetri_samkutxedi.ToString();
label2->Text = perimetri_martkutxedi.ToString();
}
```

როგორც ვხედავთ, ორის ნაცვლად ერთი ფუნქცია გვაქვს. ანუ ფუნქციების გადატვირთვის მაგივრად გამოვიყენეთ ნაგულისხმევი არგუმენტები.

ნაგულისხმევი არგუმენტები შეგვიძლია კონსტრუქტორებსაც გადავცეთ. შედეგად, შეგვიძლია თავიდან ავიცილოთ, კონსტრუქტორის გადატვირთვა მხოლოდ იმიტომ, რომ შევქმნათ როგორც ინიციალიზებული, ისე არაინიციალიზებული ობიექტები. მაგალითი:

```
//      კონსტრუქტორისთვის ნაგულისხმევი      არგუმენტის გადაცემა
class Klasi3 {
int ricxvi;
public :
        Klasi3(int par = 0) { ricxvi = par;}
        int Gacema() { return ricxvi;}
};
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi3 obj1(5);
Klasi3 obj2;

label1->Text = obj1.Gacema().ToString();
label2->Text = obj2.Gacema().ToString();
}
```

გაჩუმებით განსაზღვრული არგუმენტები შეგვიძლია მივუთითოთ ასლის კონსტრუქტორის გამოცხადებისას. მაგალითი:

```
//      გაჩუმებით განსაზღვრული არგუმენტები მითითებულია
//      ასლის კონსტრუქტორის გამოცხადებისას
class Klasi {
        int ricxvi1;
        int ricxvi2;
public :
        Klasi(int par1, int par2) {
                ricxvi1 = par1;
                ricxvi2 = par2;
        System::Windows::Forms::MessageBox::Show(L"მუშაობს ჩვეულებრივი კონსტრუქტორი");
        }
        Klasi(const Klasi& obj, int par2 = 5) {
                ricxvi1 = obj.ricxvi1;
                ricxvi2 = par2;
                System::Windows::Forms::MessageBox::Show(L"მუშაობს ასლის კონსტრუქტორი");
        }
        void Naxva() {
        System::Windows::Forms::MessageBox::Show(ricxvi1.ToString() + "   " + ricxvi2.ToString());
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Klasi obj1(ricxvi1, ricxvi2);
Klasi obj2(obj1);

obj1.Naxva();
obj2.Naxva();
}
```

ნაგულისხმევი არგუმენტები ძირითადად მაშინ გამოიყენება, როდესაც ფუნქციის გამოძახებისას პარამეტრს ხშირად ვანიჭებთ ერთსა და იმავე მნიშვნელობას.

ბოლოს, შევნიშნოთ, რომ ნაგულისხმევი არგუმენტების გამოყენება არ შეიძლება ref და value კლასებში.

გადატვირთვა და არაცალსახობა

ფუნქციის გადატვირთვისას ადგილი აქვს ***არაცალსახობას*** (ambiguity) მისი გამოძახებისას. არაცალსახობის არსი იმაში მდგომარეობს, რომ კომპილატორს არ შეუძლია გაარკვიოს თუ რომელი გადატვირთული ფუნქცია უნდა გამოიძახოს. ფუნქციის გადატვირთვისას არაცალსახობა გამოწვეულია ტიპის გარდაქმნით, პარამეტრი-მიმართვებისა და ნაგულისხმევი არგუმენტების გამოყენებით. განვიხილოთ თითოეული შემთხვევა.

მოცემული პროგრამით ხდება არაცალსახობის დემონსტრირება, რომელიც გამოწვეულია ტიპის გარდაქმნით.

```
//      არაცალსახობის დემონსტრირება,  რომელიც გამოწვეულია ტიპის გარდაქმნით
```

```
ref class Klasi       {
        public : double Funqcia(double par) {
                        return par / 2.0;
                }
                 float Funqcia(float par) {
                        return (float)(par / 4.0);
                }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj = gcnew Klasi();
float f1 = 3.5F;
double d1 = Convert::ToDouble(textBox1->Text);
int i1 = Convert::ToInt32(textBox2->Text);
//      არაცალსახობა არ არის. გამოიძახება Funqcia(float par) ფუნქცია
label1->Text = obj->Funqcia(f1).ToString();
//      არაცალსახობა არ არის. გამოიძახება Funqcia(double par) ფუნქცია
label2->Text = obj->Funqcia(d1).ToString();
//      ადგილი აქვს არაცალსახობას.
label3->Text = obj->Funqcia(i1).ToString();
}
```

აქ არაცალსახობას ადგილი აქვს obj->Funqcia(i1) ფუნქციის გამოძახებისას. ამ დროს int ტიპი შეიძლება დაყვანილი იყოს როგორც float ტიპზე, ისე double ტიპზე. ამიტომ, შეიძლება გამოძახებული იყოს როგორც obj->Funqcia(f1), ისე obj->Funqcia(d1) ფუნქცია. არაცალსახობის თავიდან ასაცილებლად Klasi კლასს უნდა დავუმატოთ მესამე ფუნქცია int Funqcia(int par) ან i1 ცვლადი გამოვაცხადოთ როგორც float ან double.

მოცემული პროგრამით ხდება არაცალსახობის დემონსტრირება, რომელიც გამოწვეულია პარამეტრი-მიმართვის გამოყენებით.

```
//      არაცალსახობის დემონსტრირება, რომელიც
//      გამოწვეულია პარამეტრი-მიმართვის გამოყენებით
ref class Klasi2 {
        public : double Funqcia(double par1) {
                        return par1 * par1;
                }
                 float Funqcia(double &par1) {
                        return par1 + par1;
                }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi2^ obj2 = gcnew Klasi2();
double wiladi = Convert::ToDouble(textBox1->Text);
//      ადგილი აქვს არაცალსახობას.
label1->Text = obj2->Funqcia(wiladi);
}
```

აქ არაცალსახობას ადგილი აქვს obj2->Funqcia(wiladi) ფუნქციის გამოძახებისას. ამ შემთხვევაში შეიძლება გამოძახებული იყოს როგორც Funqcia(double par1), ისე Funqcia(double

&par1) ფუნქცია. არაცალსახობის თავიდან ასაცილებლად ან Klasi2 კლასიდან უნდა წავშალოთ ერთ-ერთი ფუნქცია ან რომელიმე ფუნქციაში შევცვალოთ პარამეტრის ტიპი.

მოცემული მშობლიური C++ პროგრამით ხდება არაცალსახობის დემონსტრირება, რომელიც გამოწვეულია ნაგულისხმევი არგუმენტის გამოყენებით.

```
//      არაცალსახობის დემონსტრირება, რომელიც
//      გამოწვეულია ნაგულისხმევი არგუმენტის გამოყენებით
class Klasi {
        public : double Funqcia(double par) {
                        return par / 2.0;
                }
                 float Funqcia(double par1, double par2 = 15) {
                        return par1 * par2;
                }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi obj;
double wiladi1 = Convert::ToDouble(textBox1->Text);
double wiladi2 = Convert::ToDouble(textBox2->Text);
label1->Text = obj.Funqcia(wiladi1, wiladi2).ToString();
//      ადგილი აქვს არაცალსახობას.
label2->Text = obj.Funqcia(wiladi1);
}
```

აქ არაცალსახობას ადგილი აქვს obj.Funqcia(wiladi1) ფუნქციის გამოძახებისას. ამ დროს შეიძლება გამოძახებული იყოს როგორც Funqcia(double par), ისე Funqcia(double par1, double par2 = 15) ფუნქცია. არაცალსახობის თავიდან ასაცილებლად Klasi კლასის რომელიმე ფუნქციაში უნდა შევცვალოთ პარამეტრის ტიპი.

გადატვირთული ფუნქციის მისამართის მიღება

C++ ენაში შესაძლებელია გადატვირთული ფუნქციის მისამართის მიღება. *მიმთითებლის გამოცხადების გზა* განსაზღვრავს თუ გადატვირთული ფუნქციის რომელი ვერსიის მისამართი უნდა მივიღოთ. მიმთითებლის გამოცხადება შეესაბამება გადატვირთული ფუმქციების გამოცხადებას. ფუნქცია, რომლის გამოცხადებასაც შეესაბამება მიმთითებლის გამოცხადება, არის საჭირო ფუნქცია. მოცემული მშობლიური C++ პროგრამით ხდება გადატვირთული ფუნქციების მისამართების მიღება:

```
void Funqcia(int par1) {
        for ( ; par1; par1-- ) System::Windows::Forms::MessageBox::Show(par1.ToString() + "   ");
}
void Funqcia(int par1, System::Char simbolo) {
        for ( ; par1; par1-- ) System::Windows::Forms::MessageBox::Show(simbolo.ToString() + "   ");
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = ""; label2->Text = "";
int cvladi1 = Convert::ToInt32(textBox1->Text);
```

```
//        ფუნქციაზე მიმთითებლის შექმნა
void (*Funqcia1)(int);
void (*Funqcia2)(int, Char);
Funqcia1 = Funqcia;                    //    Funqcia(int) ფუნქციაზე მიმთითებლის მიღება
Funqcia2= Funqcia;                     //    Funqcia(int, Char) ფუნქციაზე მიმთითებლის მიღება
Funqcia1(cvladi1);                     //    გამოაქვს cvladi1 ცვლადის მნიშვნელობა 0-მდე
Funqcia2(cvladi1, L'ლ');               //    გამოაქვს cvladi1 რაოდენობის 'ლ' სიმბოლო*/
}
```

იმის მიხედვით, თუ როგორ არის გამოცხადებული Funqcia1 და Funqcia2 მიმთითებლები, კომპილატორი არკვევს, თუ რომელი ფუნქცია უნდა გამოიძახოს კონკრეტულ შემთხვევაში. უნდა გვახსოვდეს, რომ მიმთითებლის გამოცხადება ზუსტად უნდა შეესაბამებოდეს მხოლოდ ერთ გადატვირთულ ფუნქციას, წინააღმდეგ შემთხვევაში, ადგილი ექნება არაცალსახობას.

რეკურსია

რეკურსია არის პროცესი, როდესაც ფუნქცია თავის თავს იძახებს. ფუნქციას, რომელიც თავის თავს იძახებს - ***რეკურსიული ფუნქცია*** ეწოდება. რეკურსიული ფუნქციის საკვანძო კომპონენტი არის ოპერატორი, რომელიც ამავე ფუნქციას იძახებს.

რეკურსიის კლასიკური მაგალითია ფაქტორიალის გამოთვლა. N რიცხვის ფაქტორიალი აღინიშნება N!-ით და არის რიცხვების ნამრავლი 1-დან N-მდე. მაგალითად, 4-ის ფაქტორიალი არის 1 x 2 x 3 x 4 = 24. ქვემოთ მოცემული პროგრამით წარმოდგენილი რეკურსიული ფუნქცია განკუთვნილია რიცხვის ფაქტორიალის გამოთვლისათვის.

```
//        რეკურსიის დემონსტრირება
ref class Factorial
{
//        რეკურსიული ფუნქცია
public : int factor(int ricxvi)
{
int shedegi;
if ( ricxvi == 1 ) return 1;
shedegi = factor(ricxvi -1) * ricxvi;
return shedegi;
}
};
//
private: System::Void button13_Click(System::Object^  sender, System::EventArgs^  e) {
Factorial^ obieqti = gcnew Factorial();
int number = Convert::ToInt32(textBox1->Text);
int fact1;

label1->Text = "";
fact1 = obieqti->factor(number);
label1->Text = fact1.ToString();
}
```

რეკურსიული ფუნქციის შედგენისას სწორად უნდა განისაზღვროს რეკურსიის

დამთავრების პირობა. ამ დროს უნდა შეწყდეს ფუნქციის გამოძახება და მან უნდა დააბრუნოს მნიშვნელობა. ჩვენს შემთხვევაში, რეკურსიული გამოძახება მთავრდება მაშინ, როდესაც არგუმენტი 1-ის ტოლი ხდება. ამ დროს ფუნქცია აბრუნებს 1-ს.

რეკურსიული ფუნქციების გამოყენებას აქვს როგორც დადებითი, ისე უარყოფითი მხარეები. უარყოფითი მხარეა ის, რომ რეკურსიული პროგრამები შედარებით ნელა სრულდება. გარდა ამისა, თუ რეკურსიული გამოძახებების რაოდენობა ძალიან დიდია, მაშინ შეიძლება აღიძრას განსაკუთრებული სიტუაცია (შეცდომა). ეს ძირითადად მაშინ ხდება, როდესაც არასწორად არის განსაზღვრული რეკურსიის დამთავრების პირობა: ფუნქციის გამოძახების ნაცვლად უნდა მოხდეს მნიშვნელობის დაბრუნება.

დადებითი მხარეა ის, რომ რიგი ალგორითმებისა რეკურსიულად შეიძლება რეალიზებული იყოს უფრო ეფექტურად და მარტივად. მაგალითად, სწრაფი დახარისხების, კატალოგებში ფაილების ძებნისა და ა.შ. რეკურსიული მიდგომა ხშირად გამოიყენება, აგრეთვე, ხელოვნური ინტელექტის სფეროში.

Array კლასის ფუნქციები

Array კლასის სტატიკური Clear()ფუნქცია შეგვიძლია გამოვიყენოთ რიცხვითი მასივის ნებისმიერი მიმდევრობის გასანულებლად. მისი სინტაქსია:

Array::Clear(*მასივის_სახელი, საწყისი_ინდექსი, რაოდენობა*);

პროგრამაში Clear() ფუნქციის შესრულების შედეგად განულდება masivi მასივის ყველა ელემენტი დაწყებული ნულოვანი ელემენტიდან:

```
{
array<int>^ masivi = gcnew array<int> { 10, 20, 30, 40, 50 };
Array::Clear(masivi, 0, masivi->Length);

for each ( int cvladi in masivi )
        label1->Text += cvladi.ToString() + "  ";
}
```

პროგრამაში Clear() ფუნქციის შესრულების შედეგად განულდება masivi მასივის 3 ელემენტი დაწყებული 1 ინდექსის მქონე ელემენტიდან:

```
{
array<int>^ masivi = gcnew array<int> { 10, 20, 30, 40, 50 };
Array::Clear(masivi, 1, 3);

for each ( int cvladi in masivi )
        label1->Text += cvladi.ToString() + "  ";
}
```

თუ Clear()ფუნქციას გამოვიყენებთ String^ ტიპის კვალიანი დესკრიპტორის მიმართ, მაშინ მისი ელემენტები მიიღებენ null მნიშვნელობას. თუ ამ ფუნქციას გამოვიყენებთ bool ტიპის მასივის მიმართ, მაშინ მისი ელემენტები მიიღებენ false მნიშვნელობას.

Array კლასში განსაზღვრულია **Sort()ფუნქცია**, რომელიც ახარისხებს ერთგანზომილებიანი მასივის ელემენტებს ზრდადობით. მისი სინტაქსია:

Array::Sort(*მასივის_სახელი* [,*ინდექსი, რაოდენობა*]);

მოცემული პროგრამით ხდება Sort() ფუნქციასთან მუშაობის დემონსტრირება:

```
{
```

```
label1->Text = "";
array<int>^ masivi1 = gcnew array<int> { 10, 100, 90, 20, 30, 70, 50, 40, 80, 60 };
array<int>^ masivi2 = gcnew array<int> { 100, 90, 80, 70, 60, 50, 40, 30, 20, 10 };
//        masivi1 მასივის დახარისხება ზრდადობით
Array::Sort(masivi1);
for each ( int cvladi in masivi1 )
        label1->Text += cvladi.ToString() + "  ";

//        masivi2 მასივის ნაწილის დახარისხება ზრდადობით
Array::Sort(masivi2, 1, 4);
for each ( int cvladi in masivi2 )
        label2->Text += cvladi.ToString() + "  ";
}
```

ამ პროგრამაში მთლიანად ხდება masivi1 მასივის დახარისხება ზრდადობით, ხოლო masivi2 მასივის ნაწილობრივ დახარისხება ზრდადობით. კერძოდ, დახარისხებაში მონაწილეობს 4 ელემენტი დაწყებული 1 ინდექსის მქონე ელემენტიდან.

ქვემოთ მოცემული პროგრამით ხდება ორი მასივის ერთდროულად დახარისხება ერთი Sort() ფუნქციის გამოყენებით:

```
{
array<String^>^ saxeli = { L"ლიკა", L"რომანი", L"ანა", L"საბა" };
array<double>^ simagle = { 170.9, 185.32, 135.50, 130,70 };
//        saxeli  და saimagle მასივების დახარისხება ზრდადობით
Array::Sort(saxeli, simagle);
for ( int indexi = 0; indexi < saxeli->Length; indexi++ )
        label1->Text += saxeli[indexi]->ToString() + "  -  " + simagle[indexi].ToString() + "\n";
}
```

ერთგანზომილებიან მასივში ელემენტის მოსაძებნად გამოიყენება Array კლასის **BinarySearch() ფუნქცია**. ის გასცემს ნაპოვნი ელემენტის ინდექსს. თუ ელემენტი ვერ მოიძებნა, მაშინ გაიცემა უარყოფითი მნიშვნელობა. ამ ფუნქციის გამოყენებამდე უნდა შესრულდეს მასივის ელემენტების დახარისხება. მაგალითი:

```
{
array<int>^ masivi = { 10, 9, 57, 59, 30, 40 };
int sazebni_elementi = Convert::ToInt32(textBox1->Text);
// masivi მასივის დახარისხება
Array::Sort(masivi);
for each ( int cvladi in masivi )
        label1->Text += cvladi.ToString() + "   ";
//        ელემენტის ძებნა მასივში
int indexi = Array::BinarySearch(masivi, sazebni_elementi);
if ( indexi < 0 ) label2->Text = L"ელემენტი ვერ მოიძებნა";
        else label2->Text = L"მოიძებნა ელემენტი  -  " + sazebni_elementi.ToString() +
                                L"   მისი ინდექსია  -  " + indexi.ToString();
}
```

მოცემული პროგრამით ხდება ელემენტის ძებნა მითითებულ დიაპაზონში:

```
{
label1->Text = "";
array<int>^ masivi = { 10, 9, 57, 59, 30, 40 };
```

```
int sazebni_elementi = Convert::ToInt32(textBox1->Text);
// masivi მასივის დახარისხება
Array::Sort(masivi);
for each ( int cvladi in masivi )
        label1->Text += cvladi.ToString() + "   ";
//      ელემენტის ძებნა მითითებულ დიაპაზონში
int indexi = Array::BinarySearch(masivi,  1, 4, sazebni_elementi);
if ( indexi < 0 ) label2->Text = L"ელემენტი ვერ მოიძებნა";
        else label2->Text = L"მოიძებნა ელემენტი  -  " + sazebni_elementi.ToString() + L"   მისი
ინდექსია  -  " + indexi.ToString();
}
```

აქ ძებნა იწყება 1 ინდექსის მქონე ელემენტიდან და ძებნაში მონაწილეობს 4 ელემენტი.

საძებნი მნიშვნელობები შეიძლება მოთავსებული იყოს სტრიქონების მასივშიც. მოცემული პროგრამით სრულდება გვარების ძებნა:

```
{
int indexi;
array<String^>^ gvarebi =
        { L"ქევხიშვილი", L"ჭუმბურიძე", L"სამხარაძე", L"კაპანაძე", L"ხოშტარია", L"კირვალიძე" };
array<String^>^ sazebni_gvari = { L"კირვალიძე", L"ყალაბეგიშვილი", L"ჭუმბურიძე" };
//      gvarebi მასივის დახარისხება ზრდადობით
Array::Sort(gvarebi);
for each ( String^ cvladi in gvarebi )
        label1->Text += cvladi + "   ";
for each ( String^ cvladi in sazebni_gvari )
{
        indexi = Array::BinarySearch(gvarebi, cvladi);
if ( indexi < 0 ) label2->Text += cvladi + L"  -   ელემენტი ვერ მოიძებნა\n";
        else label2->Text += L"მოიძებნა ელემენტი  -  " + cvladi +
                                L"      მისი ინდექსია  -  " + indexi.ToString() + "\n";
}
}
```

თავი 8. მემკვიდრეობითობა. ვირტუალური ფუნქცია

მემკვიდრეობითობის საფუძვლები

როგორც ვიცით, ობიექტზე ორიენტირებული დაპროგრამების ერთ-ერთი პრინციპია მემკვიდრეობითობა. მისი გამოყენებით შეგვიძლია ახალი კლასების შექმნა, რომლებიც წარმოადგენენ საბაზო კლასის მემკვიდრე კლასებს და მემკვიდრეობით იღებენ საბაზო კლასის ყველა წევრს. მემკვიდრე კლასს დამატებული აქვს საბაზო კლასისაგან განსხვავებული წევრები. მაგალითად, შეგვიძლია შევქმნათ კლასი **ავტომობილი**, რომელსაც აქვს ისეთი საერთო მახასიათებლები, როგორიცაა ძრავის სიმძლავრე, საწვავის ხარჯი 100 კილომეტრზე, მაქსიმალური სიჩქარე. ამ კლასს შეიძლება ჰქონდეს ორი მემკვიდრე კლასი - **სატვირთო**, რომელშიც ჩნდება მახასიათებელი - ტვირთამწეობა, და **ავტობუსი**, რომელშიც ჩნდება მახასიათებელი - გადასაყვანი მგზავრების რაოდენობა.

C++ ენაში კლასს, რომლის წევრები ავტომატურად ხდება სხვა ახალი შესაქმნელი კლასის წევრები, ***წინაპარი კლასი*** (***საბაზო კლასი***, base class) ეწოდება. კლასს, რომელიც მემკვიდრეობით იღებს წინაპარი კლასის არსებულ წევრებს და მათ ახალ წევრებს უმატებს, ***მემკვიდრე კლასი*** (derived class) ეწოდება. ამრიგად, მემკვიდრე კლასი მემკვიდრეობით იღებს წინაპარ კლასში განსაზღვრულ ყველა წევრს და მათ უმატებს თავის საკუთარ წევრებს. მემკვიდრეობითობით არ გადაიცემა საბაზო კლასის კონსტრუქტორები, დესტრუქტორები და მინიჭების გადატვირთული ოპერაციები. მემკვიდრე კლასი შეიძლება იყოს წინაპარი (საბაზო) სხვა კლასებისთვის.

მემკვიდრე კლასის გამოცხადების სინტაქსია:

class ***მემკვიდრე_კლასის_სახელი*** **:** ***წინაპარი_კლასის_სახელი*** **{**
მემკვიდრე_კლასის კოდი
}

როგორც ვხედავთ, მემკვიდრე კლასის განსაზღვრისას ორი წერტილის შემდეგ ეთითება წინაპარი კლასის სახელი. ასეთი გზით შესაძლებელია მემკვიდრეობითობის იერარქიის შექმნა, რომელშიც მემკვიდრე კლასს თვითონ შეუძლია გახდეს წინაპარი სხვა კლასისთვის. არც ერთი კლასი არ შეიძლება იყოს თავისი თავის წინაპარი.

მემკვიდრეობითობის უპირატესობა იმაშია, რომ კლასის შექმნის შემდეგ, რომელიც განსაზღვრავს საერთო მახასიათებლებს, ის შეგვიძლია გამოვიყენოთ ნებისმიერი რაოდენობის მემკვიდრე კლასების შესაქმნელად, რომლებიც დამატებით შეიცავენ უნიკალურ მახასიათებლებს. შედეგად, მემკვიდრე კლასში აღარ ხდება საჭირო წინაპარ კლასში გამოცხადებული წევრების განმეორებით გამოცხადება.

განვიხილოთ პროგრამა, რომელშიც ხდება მემკვიდრეობითობის დემონსტრირება. მასში იქმნება Sibrtye საბაზო კლასი, რომელიც ინახავს გეომეტრიული ფიგურის გვერდების მნიშვნელობებს და ფუნქციას, რომელიც გასცემს მათ მნიშვნელობებს. იქმნება Sibrtye საბაზო კლასის Samkutxedi და Otxkutxedi მემკვიდრე კლასები. ამ კლასებში შესაბამისად გამოითვლება სამკუთხედისა და ოთხკუთხედის პერიმეტრი და ფართობი. რადგან, გამოიყენება String^ ტიპი, ამიტომ class სიტყვის წინ უნდა მივუთითოთ **ref** (reference, მიმართვა) საკვანძო სიტყვა.

```
//      მემკვიდრეობითობის დემონსტრირება
//      საბაზო კლასი შეიცავს გეომეტრიული ფიგურის გვერდებს
ref class Sibrtye {
public : double gverdi1;
double gverdi2;
```

```
double gverdi3;
System::String^ Naxva() {
return L"გვერდების ზომებია:  \n" +
gverdi1.ToString() + "   " + gverdi2.ToString() + "   " + gverdi3.ToString();
}
};
//        Samkutxedi კლასი არის Sibrtye კლასის მემკვიდრე
ref class Samkutxedi : public Sibrtye {
public :
System::String^ tipi;
double Partobi() {
//        Samkutxedi კლასის წევრი შეიძლება მიმართავდეს Sibrtye კლასის ღია წევრებს
return gverdi1 * gverdi2 / 2;
}
double Perimetri() {
return gverdi1 + gverdi2 + gverdi3;
}
System::String^ TipisNaxva() {
        return L"სამკუთხედის სახე - " + tipi;
}
};
//        Otxkutxedi კლასი არის Sibrtye კლასის მემკვიდრე
ref class Otxkutxedi : public Sibrtye {
public :
double Partobi() {
//        Samkutxedi კლასის წევრი შეიძლება მიმართავდეს Sibrtye კლასის ღია წევრებს
return gverdi1 * gverdi2;
}
double Perimetri() {
return gverdi1 + gverdi2 + gverdi3;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Samkutxedi^ samkutxedi = gcnew Samkutxedi();
Otxkutxedi^ oxtkutxedi = gcnew Otxkutxedi();

samkutxedi->gverdi1 = Convert::ToDouble(textBox1->Text);
samkutxedi->gverdi2 = Convert::ToDouble(textBox2->Text);
samkutxedi->gverdi3 = Convert::ToDouble(textBox3->Text);
samkutxedi->tipi = L"ტოლგვერდა";
//        ინფორმაციის გამოტანა samkutxedi ობიექტის შესახებ
label1->Text = samkutxedi->TipisNaxva();
label2->Text = samkutxedi->Naxva();
label3->Text = samkutxedi->Partobi().ToString();
label4->Text = samkutxedi->Perimetri().ToString();
```

```
oxtkutxedi->gverdi1 = Convert::ToDouble(textBox1->Text);
oxtkutxedi->gverdi2 = Convert::ToDouble(textBox2->Text);
//        ინფორმაციის გამოტანა oxtkutxedi ობიექტის შესახებ
label5->Text = oxtkutxedi->Naxva();
label6->Text = oxtkutxedi->Partobi().ToString();
label7->Text = oxtkutxedi->Perimetri().ToString();
}
```

Sibrtye კლასში განსაზღვრულია გეომეტრიული ფიგურის გვერდები. ფიგურა შეიძლება იყოს სამკუთხედი, კვადრატი, მართკუთხედი და ა.შ. Samkutxedi კლასში, რომელიც არის Sibrtye კლასის მემკვიდრე, განისაზღვრება გეომეტრიული ფიგურის კონკრეტული ტიპი, ჩვენს შემთხვევაში - სამკუთხედი. Samkutxedi კლასში ჩართულია Sibrtye კლასის ყველა წევრი და დამატებულია tipi ცვლადი, აგრეთვე, Perimetri(), Partobi() და TipisNaxva() ფუნქციები. სამკუთხედის ტიპი ინახება tipi ცვლადში, ხოლო TipisNaxva() ფუნქცია კი აბრუნებს სამკუთხედის ტიპს.

რადგან Samkutxedi კლასში ჩართულია Sibrtye წინაპარი კლასის ყველა წევრი, ამიტომ მის Partobi() ფუნქციას შეუძლია მიმართოს gverdi1 და gverdi2 ღია ცვლადებს.

მიუხედავად იმისა, რომ Sibrtye კლასი არის Samkutxedi კლასის წინაპარი, ის არის მთლიანად ავტონომიური კლასი, რომელიც შეგვიძლია დამოუკიდებლად გამოვიყენოთ. მაგალითად, სწორია შემდეგი კოდი:

```
{
Sibrtye^ Figura = gcnew Sibrtye();
Figura->gverdi1 = Convert::ToDouble(textBox1->Text);
Figura->gverdi2 = Convert::ToDouble(textBox2->Text);
label1->Text = Figura->Naxva();
}
```

Sibrtye კლასის ობიექტს არ შეუძლია მიმართოს მემკვიდრე კლასის წევრებს.

საბაზო კლასთან მიმართვის მართვა

მიმართვის სპეციფიკატორი (access specifier) მიუთითებს, თუ როგორ ხდება საბაზო კლასის წევრების გადაცემა მემკვიდრე კლასისთვის. თუ მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი, მაშინ საბაზო კლასის ყველა ღია წევრი რჩება ღია მემკვიდრე კლასშიც. თუ მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია private მოდიფიკატორი, მაშინ საბაზო კლასის ყველა ღია წევრი ხდება დახურული და შედეგად, მიუწვდომელი მემკვიდრე კლასში, თუმცა ამ წევრებთან მიმართვა შეუძლიათ მემკვიდრე კლასის ფუნქციებს. ორივე შემთხვევაში, საბაზო კლასის ყველა დახურული წევრი რჩება დახურული და შედეგად, მიუწვდომელი მემკვიდრე კლასშიც. თუ მიმართვის სპეციფიკატორი მითითებული არ არის, მაშინ იგულისხმება private.

მოცემულ პროგრამაში მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი.

```
//        საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი
ref class Sibrtye_1 {
        double gverdi3;
public : double gverdi1;
        double gverdi2;
};
```

```
//        Samkutxedi_1 კლასი არის Sibrtye_1 კლასის მემკვიდრე
ref class Samkutxedi_1 : public Sibrtye_1        {
        double fartobi;
public :
        double Fartobi() {
        fartobi = gverdi1 * gverdi2;
        return fartobi;
  }
};
//
private: System::Void button5_Click_1(System::Object^  sender, System::EventArgs^  e) {
        Samkutxedi_1^obj1 = gcnew Samkutxedi_1();
        obj1->gverdi1 = Convert::ToDouble(textBox1->Text);
        obj1->gverdi2 = Convert::ToDouble(textBox2->Text);
        double shedegi = obj1->Fartobi();
        label1->Text = shedegi.ToString();
}
```

რადგან, მემკვიდრე კლასის გამოცხადებისას, წინაპარი კლასის სახელის წინ მითითებულია public მოდიფიკატორი, ამიტომ მის ღია წევრებს შეგვიძლია მივმართოთ ძირითადი პროგრამიდან და მივანიჭოთ მნიშვნელობები. უნდა გვახსოვდეს, რომ ref და value კლასები უზრუნველყოფენ მხოლოდ ღია მემკვიდრეობითობას. ეს იმას ნიშნავს, რომ ამ პროგრამაში public მოდიფიკატორის ნაცვლად არ შეიძლება private ან protected მოდიფიკატორის მითითება.

მოცემულ პროგრამაში მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია private მოდიფიკატორი. რადგან, ვიყენებთ private მოდიფიკატორს, ამიტომ საბაზო და მემკვიდრე კლასები უნდა იყოს მშობლიური C++ ენის კლასები.

```
//        საბაზო კლასის სახელის წინ მითითებულია private მოდიფიკატორი
class Sibrtye_2 {
        double gverdi3;
public : double gverdi1;
        double gverdi2;
};
//        Samkutxedi_2 კლასი არის Sibrtye_2 კლასის მემკვიდრე
class Samkutxedi_2 : private Sibrtye_2 {
        double fartobi;
public :
void Inicializeba(double par1, double par2) {
        gverdi1 = par1;
        gverdi2 = par2;
}
double Fartobi() {
        fartobi = gverdi1 * gverdi2;
        return fartobi;
}
};
//
private: System::Void button8_Click(System::Object^  sender, System::EventArgs^  e)  {
```

```
        Samkutxedi_2  obj1;
        double cvladi1 = Convert::ToDouble(textBox1->Text);
        double cvladi2 = Convert::ToDouble(textBox2->Text);
        obj1.Inicializeba(cvladi1, cvladi2);
        double shedegi = obj1.Fartobi();
        label1->Text = shedegi.ToString();
}
```

რადგან, მემკვიდრე კლასის გამოცხადებისას, კლასის სახელის წინ მითითებულია private მოდიფიკატორი, ამიტომ მისი ღია წევრები ხდება დახურული და ძირითადი პროგრამიდან მათ ვერ მივმართავთ. მათთან მიმართვა შეუძლიათ მემკვიდრე კლასის Inicializeba და Fartobi ფუნქციებს.

protected მოდიფიკატორი

მემკვიდრეობითობის დროს ძალაშია კლასის დახურულ წევრებთან მიმართვისას არსებული შეზღუდვები. შედეგად, მემკვიდრე კლასის ფუნქციას არ შეუძლია მიმართოს წინაპარი კლასის დახურულ წევრებს. ამ პრობლემის გადასაწყვეტად გამოიყენება კლასის დაცული წევრები. ***კლასის დაცული წევრი*** ღიაა მხოლოდ მემკვიდრე კლასებისათვის და დახურულია სხვა კლასებისათვის. კლასის დაცული წევრი იქმნება protected მოდიფიკატორის საშუალებით.

თუ მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი, მაშინ საბაზო კლასის ყველა დაცული წევრი რჩება დაცული მემკვიდრე კლასშიც. თუ მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია private მოდიფიკატორი, მაშინ საბაზო კლასის ყველა დაცული წევრი ხდება დახურული. თუ მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია protected მოდიფიკატორი, მაშინ საბაზო კლასის ღია და დაცული წევრები ხდება დაცული. დაბოლოს, შევნიშნოთ, რომ protected სპეციფიკატორი შეგვიძლია გამოვიყენოთ სტრუქტურების მიმართაც.

ქვემოთ მოცემულია პროგრამა, რომელშიც გამოყენებულია protected მოდიფიკატორი. პროგრამაში მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი. როგორც ვიცით, ამ შემთხვევაში საბაზო კლასის ყველა დაცული წევრი რჩება დაცული მემკვიდრე კლასშიც.

```
//      საბაზო კლასის სახელის წინ მითითებულია public მოდიფიკატორი
ref class Sibrtye1 {
//      ეს ცვლადები ღიაა Samkutxedi კლასის წევრებისთვის
protected : int gverdi1, gverdi2, gverdi3;
public : void Inicializacia(int a, int b, int c) {
        gverdi1 = a;
        gverdi2 = b;
gverdi3 = c;
}
System::String^ Naxva() {
return gverdi1.ToString() + "   " + gverdi2.ToString() +  "   " + gverdi3.ToString();
}
};
ref class Samkutxedi1 : public Sibrtye1 {
int shedegi;
```

```
//        Partobi() ფუნქციას შეუძლია მიმართოს Sibrtye კლასში
//        განსაზღვრულ gverdi1, gverdi2 და gverdi3 ცვლადებს
public : void Partobi() {
shedegi = gverdi1 * gverdi2 / 2;
}
System::String^ PartobisNaxva() {
return shedegi.ToString();
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
//        პროგრამაში ხდება protected მოდიფიკატორის გამოყენება
Samkutxedi1^ obieqti = gcnew Samkutxedi1();
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);

//        Inicializacia() და Naxva() ფუნქციები მისაწვდომია Samkutxedi კლასის ობიექტიდან
obieqti->Inicializacia(ricxvi1, ricxvi2, ricxvi3);
label1->Text = obieqti->Naxva();
obieqti->Partobi();
label2->Text = obieqti->PartobisNaxva();
}
```

ქვემოთ მოცემულია მშობლიური C++ პროგრამა, რომელშიც მემკვიდრე კლასის გამოცხადებისას საბაზო კლასის სახელის წინ მითითებულია protected მოდიფიკატორი. როგორც ვიცით, ამ შემთხვევაში საბაზო კლასის ღია და დაცული წევრები ხდება დაცული.

```
//        საბაზო კლასის სახელის წინ მითითებულია protected მოდიფიკატორი
class Sibrtye3 {
protected : double sigane;                    //        დაცული ცვლადის გამოცხადება
//        Sibrtye კლასის კონსტრუქტორი
public :
          double simagle;
          Sibrtye3() { };
          Sibrtye3(double w, double h) {
             sigane = w;
             simagle = h;
          }
      };
//        მემკვიდრე კლასის გამოცხადება
class Samkutxedi3 : protected Sibrtye3 {
          //        წინაპარი კლასის კონსტრუქტორის გამოძახება
          public :
          Samkutxedi3(double w, double h) {
             sigane = w;
             simagle = h;
          }
          double Partobi() {
```

```
            return sigane * simagle / 2;
        }
};
//
private: System::Void button9_Click(System::Object^  sender, System::EventArgs^  e) {
double cvladi1 = Convert::ToDouble(textBox1->Text);
double cvladi2 = Convert::ToDouble(textBox2->Text);
Samkutxedi3 obj3(cvladi1, cvladi2);
double shedegi = obj3.Partobi();
label1->Text = shedegi.ToString();
}
```

კონსტრუქტორი, დესტრუქტორი და მემკვიდრეობითობა

წინაპარ და მემკვიდრე კლასს შეიძლება ჰქონდეს საკუთარი კონსტრუქტორი და დესტრუქტორი. ამ დროს წამოიჭრება კითხვა, თუ რომელი კონსტრუქტორი ქმნის მემკვიდრე კლასის ობიექტს. თუ წინაპარ კლასს და მემკვიდრე კლასს აქვთ კონსტრუქტორები და დესტრუქტორები, მაშინ კონსტრუქტორები შესრულდება მემკვიდრეობითობის მიმდევრობით, ხოლო დესტრუქტორები კი - უკუ მიმდევრობით. ამრიგად, წინაპარი კლასის კონსტრუქტორი სრულდება მემკვიდრე კლასის კონსტრუქტორზე ადრე. რაც შეეხება დესტრუქტორებს, მემკვიდრე კლასის დესტრუქტორი სრულდება წინაპარი კლასის დესტრუქტორზე ადრე.

რადგანაც, წინაპარმა კლასმა "არ იცის" მემკვიდრე კლასის არსებობის შესახებ, ამიტომ მასში ნებისმიერი ინიციალიზება სრულდება მემკვიდრე კლასისგან დამოუკიდებლად და შეიძლება გახდეს საფუძველი ნებისმიერი ინიციალიზებისათვის, რომელიც მემკვიდრე კლასში სრულდება. ამიტომ, ინიციალიზება წინაპარ კლასში უნდა შესრულდეს პირველ რიგში.

მემკვიდრე კლასის დესტრუქტორი უნდა შესრულდეს წინაპარი კლასის დესტრუქტორზე ადრე, რადგან წინაპარი კლასი საფუძვლად უდევს მემკვიდრე კლასს. ამრიგად, მემკვიდრე კლასის დესტრუქტორის გამოძახება უნდა მოხდეს მანამ, სანამ ობიექტი შეწყვეტს არსებობას.

როდესაც ინიციალიზება სრულდება მხოლოდ მემკვიდრე კლასში, მაშინ არგუმენტები გადაიცემა ჩვეულებრივი გზით. მაგრამ, როდესაც საჭირო ხდება წინაპარი კლასის კონსტრუქტორისთვის არგუმენტების გადაცემა, სიტუაცია რამდენადმე რთულდება. ჯერ, წინაპარი და მემკვიდრე კლასების ყველა საჭირო არგუმენტი გადაეცემა მემკვიდრე კლასის კონსტრუქტორს. შემდეგ, შესაბამისი არგუმენტები გადაეცემა წინაპარი კლასის კონსტრუქტორს მემკვიდრე კლასის კონსტრუქტორის საშუალებით. მემკვიდრე კლასიდან წინაპარ კლასში არგუმენტების გადაცემის სინტაქსია:

მემკვიდრე_კლასის_კონსტრუქტორი(არგუმენტების_სია) :
წინაპარი_კლასის კონსტრუქტორი(არგუმენტების_სია)

{

მემკვიდრე კლასის კონსტრუქტორის კოდი

}

წინაპარი და მემკვიდრე კლასებისთვის დასაშვებია ერთი და იგივე არგუმენტების გამოყენება. გარდა ამისა, მემკვიდრე კლასისთვის დასაშვებია ყველა არგუმენტის იგნორირება და მათი პირდაპირ გადაცემა წინაპარ კლასში.

ქვემოთ მოცემული პროგრამით ნაჩვენებია წინაპარი და მემკვიდრე კლასების კონსტრუქტორებისა და დესტრუქტორების შესრულების მიმდევრობა.

```
//        წინაპარი და მემკვიდრე კლასების კონსტრუქტორებისა
//        და დესტრუქტორების შესრულების მიმდევრობის დემონსტრირება
ref class Winapari {
public :
Winapari() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის კონსტრუქტორი");
}
~Winapari() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის დესტრუქტორი");
}
};
//        მემკვიდრე კლასი
ref class Memkvidre : public Winapari {
public :
Memkvidre() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის კონსტრუქტორი");
}
~Memkvidre() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის დესტრუქტორი");
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Memkvidre^ obj = gcnew Memkvidre();
delete obj;
}
```

როგორც პროგრამის შესრულების შედეგიდან დაინახავთ, კონსტრუქტორების გამოძახება ხდება მემკვიდრეობითობის მიმდევრობით, დესტრუქტორების გამოძახება კი - უკუ მიმდევრობით.

მოცემული პროგრამით არგუმენტი გადაეცემა მემკვიდრე კლასის კონსტრუქტორს.

```
ref class Winapari_1 {
public :
Winapari_1() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის კონსტრუქტორი");
}
~Winapari_1() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის დესტრუქტორი");
}
};
//        მემკვიდრე კლასი
ref class Memkvidre_1 : public Winapari_1 {
int cvladi;
public :
Memkvidre_1(int par) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის კონსტრუქტორი");
cvladi = par;
}
```

```
~Memkvidre_1() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის დესტრუქტორი");
}
void Naxva(System::Windows::Forms::Label^ lab1) {
lab1->Text = L"ცვლადის მნიშვნელობაა - " + cvladi.ToString();
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);
Memkvidre_1^ obj1 = gcnew Memkvidre_1(ricxvi);
obj1->Naxva(label1);
delete obj1;
}
```

თუ გვინდა რომ წინაპარი და მემკვიდრე კლასების კონსტრუქტორებს გადავცეთ ერთი ან მეტი არგუმენტი, მაშინ მემკვიდრე კლასის კონსტრუქტორს უნდა გადავცეთ ყველა არგუმენტი, რომლებიც ორივე კლასის კონსტრუქტორებს სჭირდებათ. შემდეგ, მემკვიდრე კლასის კონსტრუქტორი წინაპარი კლასის კონსტრუქტორს გადასცემს მისთვის საჭირო არგუმენტებს. ამის დემონსტრირება ხდება მოცემული პროგრამით.

```
ref class Winapari_2 {
int cvladi1;
public :
Winapari_2(int par1) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის კონსტრუქტორი");
cvladi1 = par1;
}
~Winapari_2() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს წინაპარი კლასის დესტრუქტორი");
}
void Naxva_W(System::Windows::Forms::Label^ lab1) {
lab1->Text = L"წინაპარი კლასის ცვლადის მნიშვნელობაა - " + cvladi1.ToString();
}
};
//        მემკვიდრე კლასი
ref class Memkvidre_2 : public Winapari_2 {
int cvladi2;
public :
Memkvidre_2(int par2, int par3) : Winapari_2(par3) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის კონსტრუქტორი");
cvladi2 = par2;
}
~Memkvidre_2() {
System::Windows::Forms::MessageBox::Show(L"მუშაობს მემკვიდრე კლასის დესტრუქტორი");
}
void Naxva_M(System::Windows::Forms::Label^ lab1) {
lab1->Text = L"მემკვიდრე კლასის ცვლადის მნიშვნელობაა - " + cvladi2.ToString();
}
```

```
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Memkvidre_2^ obj = gcnew Memkvidre_2(ricxvi1, ricxvi2);
obj->Naxva_M(label1);
obj->Naxva_W(label2);
delete obj;
}
```

ბოლოს შევნიშნოთ, რომ თუ წინაპარ და მემკვიდრე კლასებში გამოცხადებულია ერთი და იგივე სახელის ცვლადი, მაშინ ისინი სხვადასხვა ცვლადებად ჩაითვლება.

მრავლობითი მემკვიდრეობითობა

აქამდე ჩვენ ვიყენებდით კლასების ორდონიან იერარქიას, რომელიც შედგებოდა წინაპარი და მემკვიდრე კლასებისაგან. C++ ენაში შეგვიძლია, შევქმნათ კლასების მრავალ-დონიანი იერარქია. ამ შემთხვევაში, მემკვიდრე კლასი შეგვიძლია გამოვიყენოთ როგორც საბაზო შემდგომი მემკვიდრეობითობისათვის. მაგალითად, Klasi3 კლასი შეიძლება იყოს Klasi2 კლასის მემკვიდრე, რომელიც თავის მხრივ შეიძლება იყოს Klasi1 კლასის მემკვიდრე. ამ შემთხვევაში, Klasi3 კლასი მემკვიდრეობით იღებს Klasi1 და Klasi2 კლასების ყველა წევრს, ხოლო Klasi1 არის ირიბი (indirect) საბაზო კლასი Klasi3 კლასისთვის. Klasi3 კლასის ტიპის ობიექტის შექმნისას კონსტრუქტორები შესრულდება მემკვიდრეობითობის მიმდევრობით, ანუ ჯერ შესრულდება Klasi1 კლასის კონსტრუქტორი, შემდეგ Klasi2 კლასის კონსტრუქტორი და ბოლოს Klasi3 კლასის კონსტრუქტორი. შესაბამისად, უკუ მიმდევრობით შესრულდება დესტრუქტორები. აქვე შევნიშნოთ, რომ ნებისმიერი კლასი შეიძლება იყოს საბაზო (წინაპარი) მიუხედავად იმისა, თუ როგორ შეიქმნა ის. მოცემული პროგრამით ხდება არგუმენტების გადაცემის დემონსტრირება კლასების მრავალდონიანი იერარქიის დროს.

```
//      კლასების მრავალდონიანი იერარქიის დროს
//      არგუმენტების გადაცემის დემონსტრირება
//      Klasi1 საბაზო კლასის გამოცხადება
ref class Klasi1 {
        int cvladi1;
public :
        Klasi1(int par1) { cvladi1 = par1; }
        int Naxva1() { return cvladi1; }
};
//      Klasi2 მემკვიდრე კლასის გამოცხადება
ref class Klasi2 : public Klasi1 {
        int cvladi2;
public :
        Klasi2(int par1, int par2) : Klasi1(par2) {         // Klasi1 კლასისთვის არგუმენტების გადაცემა
                cvladi2 = par1;
        }
        int Naxva2() { return cvladi2; }
};
```

```
//        Klasi3 მემკვიდრე კლასის გამოცხადება
ref class Klasi3 : public Klasi2 {
        int cvladi3;
public :
        Klasi3(int par1, int par2, int par3) : Klasi2(par2, par3) {        // Klasi2 კლასისთვის
                cvladi3 = par1;                                            // არგუმენტების გადაცემა
        }
        int Naxva3() { return cvladi3; }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Klasi3^ obj = gcnew Klasi3(ricxvi1, ricxvi2, ricxvi3);
label1->Text = obj->Naxva1().ToString() + "  " + obj->Naxva2().ToString() +
                "   " + obj->Naxva3().ToString();
}
```

ref და value კლასებს შეიძლება მხოლოდ ერთი საბაზო კლასი ჰქონდეს. მშობლიურ C++ ენაში ერთ მემკვიდრე კლასს შეიძლება რამდენიმე წინაპარი ჰქონდეს. როდესაც კლასი უშუალოდ ხდება რამდენიმე წინაპარი კლასის მემკვიდრე, მაშინ გამოიყენება მემკვიდრე კლასის გამოცხადების შემდეგი სინტაქსი:

class *მემკვიდრე_კლასის_სახელი* **:** *მიმართვის_სპეციფიკატორი საბაზო_კლასის_სახელი1,*
მიმართვის_სპეციფიკატორი საბაზო_კლასის_სახელი2,...,
მიმართვის_სპეციფიკატორი საბაზო_კლასის_სახელიN

{
კლასის_კოდი
}

ამ შემთხვევაში, მიმართვის სპეციფიკატორმა სხვადასხვა საბაზო კლასისთვის შეიძლება სხვადასხვა მნიშვნელობა მიიღოს. კონსტრუქტორები სრულდება მარცხნიდან მარჯვნივ, იმ მიმდევრობით, რა მიმდევრობითაც არის კლასის სახელები მითითებული. შესაბამისად, დესტრუქტორები უკუ მიმდევრობით შესრულდება.

როდესაც მემკვიდრე კლასს რამდენიმე წინაპარი კლასი აქვს, რომელთა კონსტრუქტორებსაც უნდა გადაეცეს არგუმენტები, მაშინ მემკვიდრე კლასი ამ არგუმენტებს საბაზო კლასებს გადასცემს კონსტრუქტორის საშუალებით. კონსტრუქტორის გამოცხადებას შემდეგი სინტაქსი აქვს:

მემკვიდრე_კლასის_კონსტრუქტორის_სახელი (*არგუმენტების_სია*) **:**
საბაზო_კლასის_სახელი1(*არგუმენტების_სია*),
საბაზო_კლასის_სახელი2(*არგუმენტების_სია*),...,
საბაზო_კლასის_სახელიN(*არგუმენტების_სია*)

{
მემკვიდრე_კლასის_კონსტრუქტორის_კოდი
}

მემკვიდრეობითობის იერარქიის დროს მემკვიდრე კლასის კონსტრუქტორმა ყველა არგუმენტი უნდა გადასცეს წინაპარი კლასების კონსტრუქტორებს. ამის დემონსტრირება ხდება მშობლიური C++ პროგრამით.

```
//        არგუმენტების გადაცემის დემონსტრირება, როდესაც ერთ
```

```
//      მემკვიდრე კლასს ორი წინაპარი კლასი აქვს
//      Klasi5 საბაზო კლასის გამოცხადება
class Klasi4 {
        int cvladi4;
public :
        Klasi4(int par1) { cvladi4 = par1; }
        int Naxva4() { return cvladi4; }
};
//      Klasi5 საბაზო კლასის გამოცხადება
class Klasi5 {
        int cvladi5;
public :
        Klasi5(int par1)  { cvladi5 = par1; }
        int Naxva5() { return cvladi5; }
};
//      Klasi6 მემკვიდრე კლასის გამოცხადება
class Klasi6 : public Klasi4, public Klasi5 {
        int cvladi6;
public :
        Klasi6(int par1, int par2, int par3) : Klasi4(par2), Klasi5(par3) {
                cvladi6 = par1;
        }
        int Naxva6() { return cvladi6; }
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Klasi6 obj(ricxvi1, ricxvi2, ricxvi3);
label2->Text = obj.Naxva4().ToString() + "  " + obj.Naxva5().ToString() + "   " + obj.Naxva6().ToString();
}
```

ამ პროგრამაში Klasi6 კლასს ორი წინაპარი კლასი აქვს: Klasi4 და Klasi5. რადგან, ერთ კლასს ორი ან მეტი მემკვიდრე კლასი აქვს, ამიტომ ისინი უნდა იყოს მნიშვნელობითი. მიმართვით კლასს შეიძლება მხოლოდ ერთი წინაპარი კლასი ჰქონდეს.

მოცემული მშობლიური C++ პროგრამით ნაჩვენებია კონსტრუქტორების გამოძახების მიმდევრობა იმ შემთხვევაში, როდესაც ერთ მემკვიდრე კლასს რამდენიმე წინაპარი აქვს.

```
//      კონსტრუქტორების გამოძახების რიგითობის დემონსტრირება,
//      როდესაც ერთ მემკვიდრე კლასს ორი წინაპარი კლასი აქვს
//      Klasi7 საბაზო კლასის გამოცხადება
class Klasi7 {
        int cvladi7;
public :
        Klasi7() { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi7 საბაზო კლასის კონსტრუქტორი"); }
        ~Klasi7() { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi7 საბაზო კლასის დესტრუქტორი"); }
```

```
};
//      Klasi8 საბაზო კლასის გამოცხადება
class Klasi8 {
        int cvladi8;
public :
        Klasi8()  { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi8 საბაზო კლასის კონსტრუქტორი"); }
        ~Klasi8() { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi8 საბაზო კლასის დესტრუქტორი"); }
};
//      Klasi9 მემკვიდრე კლასის გამოცხადება
class Klasi9 : public Klasi7, public Klasi8 {
        int cvladi9;
public :
        Klasi9()  { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi9 საბაზო კლასის კონსტრუქტორი"); }
        ~Klasi9() { System::Windows::Forms::MessageBox::Show(
                                                L"მუშაობს Klasi9 საბაზო კლასის დესტრუქტორი"); }
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi9 obj;
}
```

მიმთითებელი მემკვიდრე კლასზე

მიმთითებლის ერთ-ერთი სპეციფიკური თავისებურება მჭიდროდაა დაკავშირებული ვირტუალურ ფუნქციასთან. კერძოდ, მიმთითებელი, რომელიც გამოცხადებულია როგორც მიმთითებელი საბაზო კლასზე, შეიძლება, აგრეთვე გამოვიყენოთ როგორც მიმთითებელი ამ საბაზო კლასის მემკვიდრე კლასზე. როდესაც მემკვიდრე კლასის ობიექტთან მიმართვისათვის ვიყენებთ საბაზო კლასზე მიმთითებელს, მაშინ შეგვიძლია მივმართოთ მემკვიდრე კლასის მხოლოდ იმ წევრებს, რომლებიც მემკვიდრეობითობით მიიღო მემკვიდრე კლასის ობიექტმა საბაზო კლასისგან. ეს იმიტომ ხდება, რომ საბაზო კლასზე მიმთითებელმა "იცის" მხოლოდ იმ წევრების შესახებ, რომლებიც მემკვიდრეობით გადაეცა მემკვიდრე კლასს და "არაფერი არ იცის" მემკვიდრე კლასის წევრების შესახებ. მაგალითი:

```
//      მემკვიდრე კლასებზე მიმთითებლების დემონსტრირება
class Sabazo {
public :
        int cvladi1;
        int cvladi2;
        int cvladi3;
        void Minicheba_S(int par1, int par2, int par3) {
                cvladi1 = par1;
                cvladi2 = par2;
                cvladi3 = par3;
        }
```

```
        int Perimetri() {
                return cvladi1 + cvladi2 + cvladi3;
        }
};
//      მემკვიდრე კლასის გამოცხადება
class Memkvidre : public Sabazo {
public :
        void Minicheba_M(int par1, int par2) {
                cvladi1 = par1;
                cvladi2 = par2;
        }
        double Fartobi() {
                return (double) ( cvladi1 + cvladi2 ) / 2;
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sabazo *mimtitebeli;                    //      საბაზო კლასზე მიმთითებელი
Sabazo obj_Sabazo;                      //      საბაზო კლასის ობიექტი
Memkvidre obj_Memkvidre;                //      მემკვიდრე კლსის ობიექტი
//      მიმთითებელს ენიჭება საბაზო კლასის ობიექტის მისამართი
mimtitebeli = &obj_Sabazo;
mimtitebeli->Minicheba_S(ricxvi1, ricxvi2, ricxvi3);
label1->Text = mimtitebeli->Perimetri().ToString();
//      მიმთითებლის საშუალებით მივმართავთ საბაზო კლასის წევრებს
mimtitebeli = &obj_Memkvidre;
mimtitebeli->Minicheba_S(ricxvi1, ricxvi2, ricxvi3);
//      მიმთითებლის საშუალებით ვერ მივმართავთ მემკვიდრე კლასის წევრებს
obj_Memkvidre.Minicheba_M(ricxvi1, ricxvi2);
label2->Text = obj_Memkvidre.Fartobi().ToString();
}
```

იგივე პროგრამის C++/CLI ვერსიას აქვს სახე:

```
ref class Sabazo {
public :
        int cvladi1;
        int cvladi2;
        int cvladi3;
        void Minicheba_S(int par1, int par2, int par3) {
                cvladi1 = par1;
                cvladi2 = par2;
                cvladi3 = par3;
        }
        int Perimetri() {
                return cvladi1 + cvladi2 + cvladi3;
        }
```

```
};
//      მემკვიდრე კლასის გამოცხადება
ref class Memkvidre : public Sabazo {
public :
        void Minicheba_M(int par1, int par2) {
                cvladi1 = par1;
                cvladi2 = par2;
        }
        double Fartobi() {
                return (double) ( cvladi1 + cvladi2 ) / 2;
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sabazo^ mimtitebeli;                                    //      საბაზო კლასზე მიმთითებელი
Sabazo^ obj_Sabazo = gcnew Sabazo();                    //      საბაზო კლასის ობიექტი
Memkvidre^ obj_Memkvidre = gcnew Memkvidre();           //      მემკვიდრე კლსის ობიექტი
//      მიმთითებელს ენიჭება საბაზო კლასის ობიექტის მისამართი
mimtitebeli = obj_Sabazo;
mimtitebeli->Minicheba_S(ricxvi1, ricxvi2, ricxvi3);
label1->Text = mimtitebeli->Perimetri().ToString();
//      მიმთითებლის საშუალებით მივმართავთ საბაზო კლასის წევრებს
mimtitebeli = obj_Memkvidre;
mimtitebeli->Minicheba_S(ricxvi1, ricxvi2, ricxvi3);
//      მიმთითებლის საშუალებით ვერ მივმართავთ მემკვიდრე კლასის წევრებს
obj_Memkvidre->Minicheba_M(ricxvi1, ricxvi2);
label2->Text = obj_Memkvidre->Fartobi().ToString();
}
```

საბაზო კლასზე მიმთითებელი შეიძლება გამოვიყონოთ მემკვიდრე კლასის ობიექტთან მიმართვისათვის. მაგრამ, მემკვიდრე კლასზე მიმთითებელი არ შეიძლება გამოვიყენოთ საბაზო კლასის ობიექტთან მიმართვისათვის. ბოლოს შევნიშნოთ, რომ თუ საბაზო კლასზე მიმთითებელი მიმართავს მემკვიდრე კლასის ობიექტს და მისი მნიშვნელობა ერთით გაიზარდა, მაშინ ის მიმართავს არა მემკვიდრე კლასის მომდევნო ობიექტს, არამედ საბაზო კლასის მომდევნო ობიექტს.

ვირტუალური ფუნქცია

C++ ენაში ვირტუალური ფუნქცია უზრუნველყოფს პოლიმორფიზმს პროგრამის შესრულებისას. პოლიმორფიზმი საშუალებას იძლევა წინაპარ კლასში განვსაზღვროთ ფუნქცია, რომელიც საერთო იქნება ყველა მემკვიდრე კლასისათვის. ამასთან, ამ ფუნქციის კოდი მემკვიდრე კლასში შეიძლება შეიცვალოს ან იგივე დარჩეს. უცვლელი რჩება ფუნქციის სახელი და მიმართვითი ცვლადი. ფუნქციების ხელახალი განსაზღვრა არის პოლიმორფიზმის "ერთი ინტერფეისი - რამდენიმე ფუნქცია" პრინციპის რეალიზების ერთ-ერთი საშუალება.

ფუნქციას, რომლის განსაზღვრისას წინაპარ კლასში მითითებული იყო virtual საკვანძო

სიტყვა, და რომელიც ხელახლა იყო განსაზღვრული ერთ ან მეტ მემკვიდრე კლასში, ***ვირტუალური ფუნქცია*** ეწოდება. თითოეულ მემკვიდრე კლასს შეიძლება ჰქონდეს ვირტუალური ფუნქციის საკუთარი ვერსია. C++ ენაში ვირტუალური ფუნქციის ვერსია აირჩევა იმ ობიექტის ტიპის შესაბამისად, რომელსაც მიმთითებელი მიმართავს. ეს არჩევა ხორციელდება პროგრამის შესრულებისას. მიმთითებელი შეიძლება მიმართავდეს სხვადასხვა ტიპის ობიექტებს, ამიტომ, შეიძლება გამოძახებული იყოს ვირტუალური ფუნქციების სხვადასხვა ვერსია. სხვა სიტყვებით რომ ვთქვათ, სწორედ ობიექტის ტიპი, რომელზეც მიუთითებს მიმთითებელი, განსაზღვრავს გამოსაძახებელ ვირტუალურ ფუნქციას.

თუ საბაზო კლასზე მიმთითებელი მიმართავს მემკვიდრე კლასის ობიექტს, რომელიც ვირტუალურ ფუნქციას შეიცავს, მაშინ ობიექტის ტიპის მიხედვით, რომელსაც მიმთითებელი მიმართავს, განისაზღვრება თუ რომელი ვირტუალური ფუნქციის გამოძახება უნდა შესრულდეს. ამასთან, გამოსაძახებელი ვირტუალური ფუნქცია განისაზღვრება პროგრამის შესრულებისას. ამრიგად, გამოსაძახებელი ვირტუალური ფუნქცია განსაზღვრავს იმ ობიექტის ტიპს, რომელსაც მიმთითებელი მიმართავს. გამოსაძახებელი ვირტუალური ფუნქციის დინამიკურად არჩევის პროცესი არის ***დინამიკური პოლიმორფიზმის*** პრინციპის რეალიზება. ვირტუალური ფუნქციის შემცველ კლასს ***პოლიმორფული კლასი*** (polymorphic class) ეწოდება.

წინაპარ კლასში ვირტუალური ფუნქციის განსაზღვრისას დასაბრუნებელი მნიშვნელობის ტიპის წინ ეთითება virtual საკვანძო სიტყვა, ხოლო მემკვიდრე კლასში ვირტუალური ფუნქციის ხელახალი განსაზღვრისას virtual სიტყვის მითითება საჭირო არ არის. მემკვიდრე კლასში ვირტუალური ფუნქციის განსაზღვრის პროცესს, როდესაც ნაწილობრივ ან მთლიანად იცვლება ფუნქციის ტანი, ხოლო ფუნქციის სახელი, პარამეტრები და მათი ტიპები იგივე რჩება, ***ფუნქციის ხელახალი განსაზღვრა*** ეწოდება.

ფუნქციის ხელახალი განსაზღვრა საფუძვლად უდევს გამოსაძახებელი ფუნქციის დინამიკურად არჩევის კონცეფციას. ეს არის მექანიზმი, რომლის საშუალებითაც გამოსაძახებელი ხელახლა განსაზღვრული ფუნქციის არჩევა ხორციელდება პროგრამის შესრულების და არა პროგრამის კომპილირებისას.

ვირტუალური ფუნქცია გამოიძახება ჩვეულებრივი ფუნქციის მსგავსად. მაგრამ, დინამიკური პოლიმორფიზმის უზრუნველყოფა ხდება მაშინ, როდესაც ვირტუალური ფუნქცია გამოიძახება მიმთითებლის საშუალებით.

ქვემოთ მოცემულია პროგრამა, რომელშიც ნაჩვენებია ვირტუალური ფუნქციისა და მისი ხელახლა განსაზღვრული ვერსიების გამოყენება.

```
//      ვირტუალურ ფუნქციებთან მუშაობის დემონსტრირება
ref class Sibrtye {
        int gverdi1, gverdi2, gverdi3;
public :
        void Minicheba(int par1, int par2, int par3) {
                gverdi1 = par1;
                gverdi2 = par2;
                gverdi3 = par3;
        }
        void Gacema(int &par1, int &par2, int &par3) {
                par1 = gverdi1;
                par2 = gverdi2;
                par3 = gverdi3;
        }
        virtual int Perimetri() {
                return 0;
        }
```

```
};
//        მემკვიდრე კლასის გამოცხადება
ref class Samkutxedi : public Sibrtye {
public :
virtual  int Perimetri() override {
                    int cvladi1, cvladi2, cvladi3;
                    Gacema(cvladi1, cvladi2, cvladi3);
                    return cvladi1 + cvladi2 + cvladi3;
          }
};
//        მემკვიდრე კლასის გამოცხადება
ref class Martkutxedi : public Sibrtye {
public :
virtual  int Perimetri() override {
                    int cvladi1, cvladi2, cvladi3;
                    Gacema(cvladi1, cvladi2, cvladi3);
                    return ( cvladi1 + cvladi2 ) * 2;
          }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sibrtye^ mimtitebeli;
Samkutxedi^ obj_Sam = gcnew Samkutxedi();
Martkutxedi^ obj_Mart = gcnew Martkutxedi();

obj_Sam->Minicheba(ricxvi1, ricxvi2, ricxvi3);
obj_Mart->Minicheba(ricxvi1, ricxvi2, ricxvi3);
mimtitebeli = obj_Sam;
label1->Text = mimtitebeli->Perimetri().ToString();
mimtitebeli = obj_Mart;
label2->Text = mimtitebeli->Perimetri().ToString();
}
```

როგორც ვხედავთ, Sibrtye კლასში ვირტუალური ფუნქციის კოდი არ არის განსაზღვრული, რადგან ამ კლასში მის განსაზღვრას აზრი არ აქვს. სამაგიეროდ ის ხელახლაა განსაზღვრული მემკვიდრე კლასებში, რომლებშიც მის განსაზღვრას აზრი აქვს.

ფუნქციის გადატვირთვა და ხელახალი განსაზღვრა ერთმანეთისაგან განსხვავებული პროცესებია:

1. გადატვირთული ფუნქციები ერთმანეთისაგან უნდა განსხვავდებოდეს პარამეტრების რაოდენობით და ტიპებით. ხალახლა განსაზღვრულ ვირტუალურ ფუნქციებს უნდა ჰქონდეს პარამეტრების ერთნაირი რაოდენობა, პარამეტრებისა და დასაბრუნებელი მნიშვნელობების ერთნაირი ტიპები.
2. ვირტუალური ფუნქცია უნდა იყოს კლასის წევრი. გადატვირთული ფუნქცია შეიძლება იყოს ან არ იყოს კლასის წევრი.
3. დესტრუქტორები შეიძლება იყოს ვირტუალური, კონსტრუქტორები კი - არა.

ფუნქციების გადატვირთვასა და ხელახალ განსაზღვრას შორის განსხვავების აღსანიშნავად მემკვიდრე კლასის ვირტუალური ფუნქციის გამოცხადებისას გამოიყენება ტერმინი - ***შეცვლა*** (override).

ვირტუალურ ფუნქციებს ახასიათებს მემკვიდრეობითობის იერარქიული სტრუქტურა. თუ ვირტუალური ფუნქციის ვერსია არ არის განსაზღვრული მემკვიდრე კლასში, მაშინ გამოიყენება ვირტუალური ფუნქციის საბაზო კლასში განსაზღვრული ვერსია. ამის დემონსტრირება ხდება მოცემული პროგრამით.

```
//      ვირტუალურ ფუნქციებთან მუშაობის დემონსტრირება
ref class Sabazo {
public :
        int cvladi;
        Sabazo(int par) { cvladi = par; }
        virtual void Funqcia() {
System::Windows::Forms::MessageBox::Show(
L"მუშაობს წინაპარი კლასის ვირტუალური ფუნქცია");
        }
};
//      პირველი მემკვიდრე კლასის გამოცხადება
ref class Memkvidre1 : public Sabazo {
public : Memkvidre1(int par) : Sabazo(par) {  }
virtual  void Funqcia() override {
System::Windows::Forms::MessageBox::Show(
L"მუშაობს პირველი მემკვიდრე კლასის ვირტუალური ფუნქცია");
        }
};
//      მეორე მემკვიდრე კლასის გამოცხადება
ref class Memkvidre2 : public Sabazo {
public : Memkvidre2(int par) : Sabazo(par) {  }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sabazo^ mimtitebeli;
Sabazo^ obj_W = gcnew Sabazo(ricxvi1);
Memkvidre1^ obj_M1 = gcnew Memkvidre1(ricxvi2);
Memkvidre2^ obj_M2 = gcnew Memkvidre2(ricxvi3);

mimtitebeli = obj_W;
mimtitebeli->Funqcia();                 //      გამოიძახება Sabazo კლასის Funqcia() ფუნქცია

mimtitebeli = obj_M1;
mimtitebeli->Funqcia();                 //      გამოიძახება Memkvidre1 კლასის Funqcia() ფუნქცია

mimtitebeli = obj_M2;
mimtitebeli->Funqcia();                 //      გამოიძახება Sabazo კლასის Funqcia() ფუნქცია
```

}

როგორც ვხედავთ, მეორე მემკვიდრე კლასში გამოცხადებული არ არის ვირტუალური ფუნქცია. ამიტომ, ამ კლასის ობიექტთან მიმართვისას იმუშავებს წინაპარ კლასში განსაზღვრული ვირტუალური ფუნქცია.

თუ საბაზო კლასში განსაზღვრულია ვირტუალური ფუნქცია, მაშინ ის შეიძლება ხელახლა იყოს განსაზღვრული ყველა ან ზოგიერთ მემკვიდრე კლასში. დავუშვათ, Sabazo საბაზო კლასში განსაზღვრულია ვირტუალური ფუნქცია და Memkvidre1 არის Sabazo კლასის მემკვიდრე, ხოლო Memkvidre2 არის Memkvidre1 კლასის მემკვიდრე. თუ Memkvidre1 კლასში ხელახლა განსაზღვრული არ არის ვირტუალური ფუნქცია, მაშინ ამ კლასის ობიექტთან მიმართვისას გამოყენებული იქნება Sabazo კლასში განსაზღვრული ვირტუალური ფუნქცია. თუ Memkvidre1 კლასში ხელახლაა განსაზღვრული ვირტუალური ფუნქცია და არ არის განსაზღვრული Memkvidre2 კლასში, მაშინ Memkvidre2 კლასის ობიექტთან მიმართვისას გამოყენებული იქნება Memkvidre1 კლასში განსაზღვრული ვირტუალური ფუნქცია. მოცემული პროგრამით ხდება ყოველივე ამის დემონსტრირება.

```
//
ref class Sabazo {
public :
        int cvladi;
        Sabazo(int par) { cvladi = par; }
        virtual void Funqcia() {
                        System::Windows::Forms::MessageBox::Show(
                        L"მუშაობს წინაპარი კლასის ვირტუალური ფუნქცია");
}
};
//      პირველი მემკვიდრე კლასის გამოცხადება
ref class Memkvidre1 : public Sabazo {
public : Memkvidre1(int par) : Sabazo(par) {  }
virtual  void Funqcia() override {
System::Windows::Forms::MessageBox::Show(
L"მუშაობს პირველი მემკვიდრე კლასის ვირტუალური ფუნქცია");
}
};
//      მეორე მემკვიდრე კლასის გამოცხადება
ref class Memkvidre2 : public Memkvidre1 {
public : Memkvidre2(int par) : Memkvidre1(par) {  }
};
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sabazo^ mimtitebeli;
Sabazo^ obj_W = gcnew Sabazo(ricxvi1);
Memkvidre1^ obj_M1 = gcnew Memkvidre1(ricxvi2);
Memkvidre2^ obj_M2 = gcnew Memkvidre2(ricxvi3);

mimtitebeli = obj_W;
mimtitebeli->Funqcia();
```

```
mimtitebeli = obj_M1;
mimtitebeli->Funqcia();

mimtitebeli = obj_M2;
mimtitebeli->Funqcia();
}
```

სუფთა ვირტუალური ფუნქცია

ზოგჯერ საჭიროა ისეთი წინაპარი კლასის შექმნა, სადაც უნდა განისაზღვროს ფუნქციები, რომელთა მიერ მემკვიდრე კლასის ობიექტებში შესრულებული მოქმედებები უცნობია. კლასში ასეთი ფუნქციებისთვის განისაზღვრება მხოლოდ პროტოტიპი, კოდი კი ცარიელია. კლასში გამოცხადებულ ვირტუალურ ფუნქციას, რომელიც არანაირ მოქმედებას არ ასრულებს, ***სუფთა ვირტუალური ფუნქცია*** (pure virtual function) ეწოდება. ის აუცილებლად უნდა იყოს ხელახლა განსაზღვრული მემკვიდრე კლასში. საბაზო კლასში მოთავსებულია მხოლოდ სუფთა ვირტუალური ფუნქციის პროტოტიპი. სუფთა ვირტუალური ფუნქციის გამოცხადების სინტაქსია:

virtual *ტიპი ფუნქციის_სახელი* (*პარამეტრების_სია*) = 0;

ფუნქციის ნულთან ტოლობა ნიშნავს იმას, რომ საბაზო კლასში არ არის განსაზღვრული ფუნქციის კოდი. მოცემული პროგრამით ხდება სუფთა ვირტუალურ ფუნქციასთან მუშაობის დემონსტრირება.

```
//      სუფთა ვირტუალურ ფუნქციასთან მუშაობის დემონსტრირება
ref class Sibrtye1  {
        int gverdi1, gverdi2, gverdi3;
public :
        void Minicheba(int par1, int par2, int par3) {
                gverdi1 = par1;
                gverdi2 = par2;
                gverdi3 = par3;
        }
        void Gacema(int &par1, int &par2, int &par3) {
                par1 = gverdi1;
                par2 = gverdi2;
                par3 = gverdi3;
        }
        virtual int Perimetri() = 0;                //      სუფთა ვირტუალური ფუნქცია
};
//      მემკვიდრე კლასის გამოცხადება
ref class Samkutxedi1 : public Sibrtye1 {
public :
virtual int Perimetri() override {
        int cvladi1, cvladi2, cvladi3;
        Gacema(cvladi1, cvladi2, cvladi3);
        return cvladi1 + cvladi2 + cvladi3;
}
};
//      მემკვიდრე კლასის გამოცხადება
```

```
ref class Martkutxedi1 : public Sibrtye1 {
public :
virtual int Perimetri() override {
        int cvladi1, cvladi2, cvladi3;
        Gacema(cvladi1, cvladi2, cvladi3);
        return ( cvladi1 + cvladi2 ) * 2;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Sibrtye1^ mimtitebeli;
Samkutxedi1^ obj_Sam = gcnew Samkutxedi1();
Martkutxedi1^ obj_Mart = gcnew Martkutxedi1();

obj_Sam->Minicheba(ricxvi1, ricxvi2, ricxvi3);
obj_Mart->Minicheba(ricxvi1, ricxvi2, ricxvi3);
mimtitebeli = obj_Sam;
label1->Text = mimtitebeli->Perimetri().ToString();
mimtitebeli = obj_Mart;
label2->Text = mimtitebeli->Perimetri().ToString();
}
```

აბსტრაქტული კლასი

კლასს, რომელიც სუფთა ვირტუალურ ფუნქციას შეიცავს, ***აბსტრაქტული კლასი*** (abstract class) ეწოდება. აბსტრაქტული კლასი შეიძლება იყოს მხოლოდ საბაზო.

აბსტრაქტული კლასის არსებობის აუცილებლობა გამოწვეულია კლასში ისეთი ფუნქციების ჩართვის აუცილებლობით, რომლებისთვისაც განსაზღვრულია მხოლოდ ფუნქციის ხელმოწერა, მაგრამ უცნობია ფუნქციის მიერ შესრულებული მოქმედებები ანუ ფუნქციის კოდი. ასეთი ფუნქციების რეალიზება ანუ კოდის განსაზღვრა მემკვიდრე კლასებში უნდა მოხდეს.

რადგან, ფუნქციის კოდი განსაზღვრული არ არის, ამიტომ აბსტრაქტული კლასის ობიექტის შექმნა შეუძლებელია. ვირტუალური ფუნქცია შეგვიძლია გამოვაცხადოთ, აგრეთვე როგორც აბსტრაქტული. აბსტრაქტულია, აგრეთვე, კლასი თუ ის აბსტრაქტულ ფუნქციას შეიცავს. აბსტრაქტული ფუნქციის გამოცხადების სინტაქსია:

virtual *გაცემული_ტიპი* ფუნქციის_*სახელი*(*პარამეტრების_სია*) abstract;

თუ კლასი შეიცავს აბსტრაქტულ ფუნქციას, მაშინ კლასის სახელის შემდეგ უნდა მივუთითოთ abstract მოდიფიკატორი.

abstract მოდიფიკატორი შეგვიძლია გამოვიყენოთ მხოლოდ ჩვეულებრივი მეთოდის მიმართ და არ შეიძლება გამოვიყენოთ სტატიკური მეთოდის მიმართ.

კლასი, რომელიც ერთ ან მეტ აბსტრაქტულ ფუნქციას შეიცავს, გამოცხადებული უნდა იყოს როგორც აბსტრაქტული, ე.ი. მისი გამოცხადებისას უნდა მივუთითოთ abstract მოდიფიკატორი. რადგან, აბსტრაქტული კლასი არ არის განსაზღვრული ბოლომდე, ამიტომ

შეუძლებელია ამ კლასის ობიექტის შექმნა.

ქვემოთ მოცემულ პროგრამაში Sibrtye კლასი გამოცხადებულია როგორც აბსტრაქტული. მასში გამოცხადებულია აბსტრაქტული Partobi() ფუნქცია, რომელიც განკუთვნილია ორგანზომილებიანი გეომეტრიული ფიგურის ფართობის გამოსათვლელად. ამიტომ, Sibrtye კლასის მემკვიდრე კლასების შექმნისას მათში ხელახლა უნდა იყოს განსაზღვრული Partobi() ფუნქცია.

```
//      აბსტრაქტული საბაზო კლასი
ref class Sabazo abstract {
public :
//
virtual double Funqcia() abstract;
//
virtual void Naxva()
{
        System::Windows::Forms::MessageBox::Show(L"მოცულობა = " + Funqcia().ToString());
}
};
//
ref class Memkvidre_1 : Sabazo {
public :
//
virtual void Naxva() override
{
System::Windows::Forms::MessageBox::Show(L"სასარგებლო მოცულობა = " + Funqcia().ToString());
}
//
virtual double Funqcia() override
{
        return ricxvi1 + ricxvi2 + ricxvi3;
}
//
Memkvidre_1() : ricxvi1(10.5), ricxvi2(15.8), ricxvi3(20.4) { }
//
Memkvidre_1(double par1, double par2, double par3) : ricxvi1(par1), ricxvi2(par2), ricxvi3(par3) { }
protected :
double ricxvi1;
double ricxvi2;
double ricxvi3;
};
//
ref class Memkvidre_2 : Memkvidre_1 {
public :
//
virtual double Funqcia() override
{
        return ricxvi1 * ricxvi2 * ricxvi3;
}
```

```
//
        Memkvidre_2(double par1, double par2, double par3) : Memkvidre_1(par1, par2, par3) { }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
array<Memkvidre_1^>^ masivi =
                        { gcnew Memkvidre_1(2.0, 3.0, 4.0), gcnew Memkvidre_2(2.0, 3.0, 4.0),
                          gcnew Memkvidre_1(4.0, 5.0, 6.0), gcnew Memkvidre_2(4.0, 5.0, 6.0)  };
for each ( Memkvidre_1^ memk_obj in masivi )
        memk_obj->Naxva();
}
```

აქ Sabazo კლასის სახელს მოსდევს abstract საკვანძო სიტყვა. თუ C++/CLI კლასი შეიცავს მშობლიური C++ ენის სუფთა ვირტუალური ფუნქციის ეკვივალენტს, მაშინ უნდა განვსაზღვროთ აბსტრაქტული ფუნქციები, რაც თავიდან აგვაცილებს ამ კლასის ტიპის ობიექტების შექმნას. abstract საკვანძო სიტყვა ჩაწერილია, აგრეთვე Funqcia() ფუნქციის გამოცხადებისას იმის მისათითებლად, რომ ის განსაზღვრული უნდა იყოს ამ კლასისთვისაც.

როდესაც კლასი არის აბსტრაქტული კლასის მემკვიდრე, მან უნდა მოახდინოს წინაპარი კლასის ყველა აბსტრაქტული ფუნქციის რეალიზება. წინააღმდეგ შემთხვევაში, ასეთი მემკვიდრე კლასი გამოცხადებული უნდა იყოს abstract მოდიფიკატორით. ამრიგად, abstract ატრიბუტი მემკვიდრეობით გადაიცემა მანამ, სანამ მეთოდები და - ე.ი. თვით კლასი, არ იქნება მთლიანად რეალიზებული.

საბაზო კლასი მიმართვითი კლასისთვის ყოველთვის ღიაა და public საკვანძო სიტყვის მითითება აუცილებელი არ არის. რადგან, პარამეტრებისთვის ვერ გამოვიყენებთ ნაგულისხმევ მნიშვნელობებს, ამიტომ უნდა განვსაზღვროთ უარგუმენტებო კონსტრუქტორი, რომელიც შეასრულებს ricxvi1, ricxvi2 და ricxvi3 ველების ინიციალიზებას. Memkvidre_1 კლასში Funqcia() ფუნქცია ხელახლაა განსაზღვრული. ყოველთვის უნდა გამოვიყენოთ override საკვანძო სიტყვა, როდესაც ვაპირებთ საბაზო კლასის ფუნქციის ხელახლა განსაზღვრას. თუ Memkvidre_1 კლასში Funqcia() ფუნქცია არ არის რეალიზებული, ის იქნება აბსტრაქტული.

ძირითად პროგრამაში იქმნება Memkvidre_1 დესკრიპტორების მასივი. რადგან Memkvidre_1 და Memkvidre_2 მიმართვითი კლასებია, ამიტომ ობიექტები იქმნება CLR გროვაში. ობიექტების მიღებული მისამართები ახდენენ masivi მასივის ელემენტების ინიციალიზებას. იქმნება stack1 ობიექტი და მასში ხდება Memkvidre_1 და Memkvidre_2 ობიექტების მოთავსება.

მშობლიური C++ ენისაგან განსხვავებით:

- მხოლოდ მიმართვითი კლასები შეიძლება იყოს მემკვიდრეობითობით მიღებული კლასების ტიპები.
- საბაზო კლასი მემკვიდრე მიმართვითი კლასისთვის ყოველთვის არის public.
- ფუნქცია, რომელიც განსაზღვრული არ არის მიმართვით კლასში, არის აბსტრაქტული და გამოცხადებული უნდა იყოს abstract საკვანძო სიტყვის გამოყენებით.
- კლასი, რომელიც შეიცავს ერთ ან მეტ აბსტრაქტულ ფუნქციას, უნდა იყოს განსაზღვრული როგორც აბსტრაქტული abstract საკვანძო სიტყვის გამოყენებით.
- კლასი, რომელიც არ შეიცავს აბსტრაქტულ ფუნქციებს, შეიძლება იყოს განსაზღვრული როგორც აბსტრაქტული. ამ შემთხვევაში, ასეთი კლასის ეგზემპლარები არ შეიქმნება.
- ჩვენ აშკარად უნდა გამოვიყენოთ override საკვანძო სიტყვა ფუნქციის ხელახალი განსაზღვრისათვის.

საბაზო კლასის ფუნქციის დამალვა მემკვიდრეობითობის დროს

მემკვიდრე კლასში ფუნქცია შეგვიძლია განვსაზღვროთ როგორც new. ამ შემთხვევაში ის მალავს ასეთივე სიგნატურის მქონე საბაზო კლასის ფუნქციას და მემკვიდრე კლასის ფუნქცია აღარ მიიღებს მონაწილეობას პოლიმორფულ ქცევაში. მოცემული პროგრამით ხდება ამის დემონსტრირება.

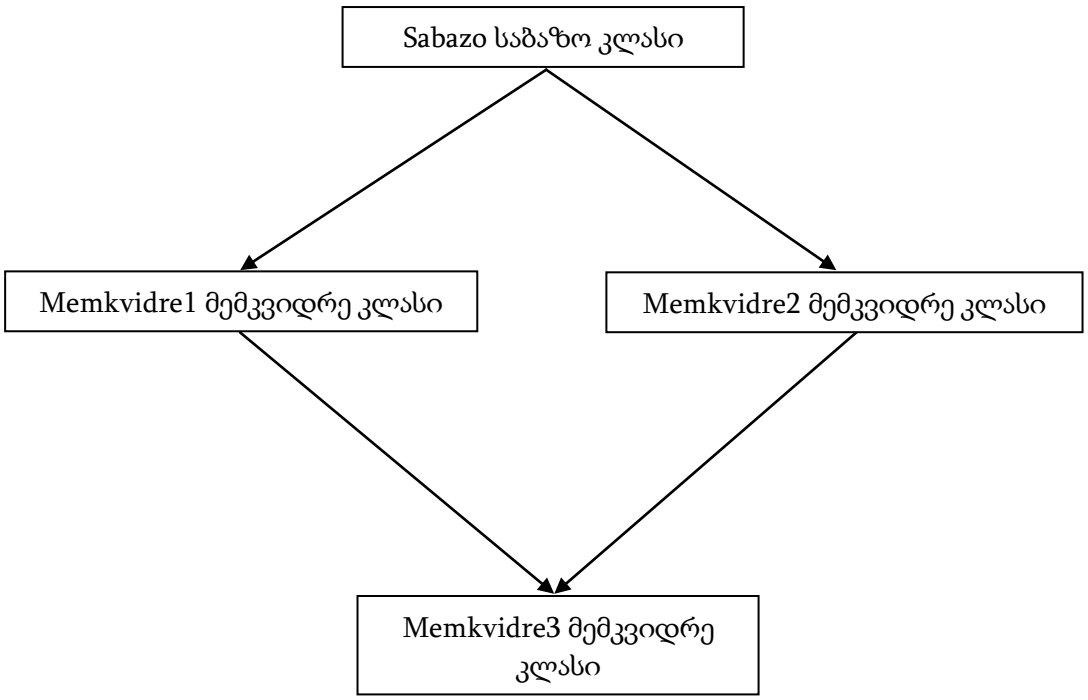
```
ref class SabazoKlasi {
public :
int i;
SabazoKlasi() {}
SabazoKlasi(int par1) {
i = par1;
};
void Naxva(System::Windows::Forms::Label^ lab1) {
i += 10;
lab1->Text = L"მუშაობს საბაზო კლასის ფუნქცია.   i = " + i.ToString() + "\n";
}
};
//          მემკვიდრე კლასი
ref class MemkvidreKlasi : SabazoKlasi {
public :
MemkvidreKlasi(int par2) : SabazoKlasi(par2) { }
//          ეს Naxva() ფუნქცია მალავს SabazoKlasi კლასში გამოცხადებულ Naxva() ფუნქციას
virtual void Naxva(System::Windows::Forms::Label^ lab1) new
{
i *= 2;
lab1->Text += L"მუშაობს მემკვიდრე კლასის ფუნქცია.   i = " + i.ToString() + "\n";
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
SabazoKlasi^ Sabazo_obj = gcnew MemkvidreKlasi(ricxvi2);
MemkvidreKlasi^ memkvidre_obj = gcnew MemkvidreKlasi(ricxvi1);
memkvidre_obj->Naxva(label1);
Sabazo_obj->Naxva(label2);
}
```

MemkvidreKlasi კლასში გამოცხადებული Naxva() ფუნქცია მალავს SabazoKlasi კლასში გამოცხადებულ Naxva() ფუნქციას, რადგან მისი გამოცხადებისას მითითებულია new საკვანძო სიტყვა. ამის დემონსტრირებას ახდენს პროგრამის memkvidre_obj->Naxva(label1); სტრიქონი.

რადგან მემკვიდრე კლასში გამოცხადებული Naxva() ფუნქცია არ არის პოლიმორფული, ამიტომ საბაზო კლასის ტიპის დესკრიპტორის გამოყენებით ამ ფუნქციას ვერ გამოვიძახებთ. ამის დემონსტრირებას ახდენს პროგრამის Sabazo_obj->Naxva(label2); სტრიქონი.

ვირტუალური საბაზო კლასი

კლასების მრავალდონიანი იერარქიის შექმნისას შეიძლება გაჩნდეს შემდეგი პრობლემა. დავუშათ, გვაქვს Sabazo კლასი, რომელსაც ორი Memkvidre1 და Memkvidre2 კლასები აქვს (ნახ. 7.1). ორივე ეს მემკვიდრე კლასი არის წინაპარი Memkvidre3 კლასისთვის. ამ შემთხვევაში, Sabazo კლასის წევრები ორჯერ გადაეცემა Memkvidre3 კლასს: ერთხელ Memkvidre1 კლასის საშუალებით და მეორედ Memkvidre2 კლასის საშუალებით. არაცალსახობა გვექნება მაშინ, როდესაც Sabazo კლასის წევრი გამოყენებული იქნება Memkvidre3 კლასში. საქმე ის არის, რომ Memkvidre3 კლასში იარსებებს Sabazo კლასის ორი ასლი, რომლებიც მიიღება Memkvidre1 და Memkvidre2 კლასებისაგან და შეუძლებელი იქნება იმის გარკვევა თუ რომელ ასლთან უნდა შესრულდეს მიმართვა. ამ პრობლემის გადასაჭრელად გამოიყენება ***ვირტუალური საბაზო კლასი*** (virtual base class). ამ შემთხვევაში მემკვიდრე კლასის გამოცხადებისას, საბაზო კლასის სახელის წინ უნდა მივუთითოთ virtual საკვანძო სიტყვა. აქვე შევნიშნოთ, რომ ref და value კლასები არ შეიძლება იყოს ვირტუალური საბაზო კლასები.



ნახ. 7.1.

მოცემული მშობლიური C++ პროგრამით ხდება ვირტუალურ საბაზო კლასთან მუშაობის დემონსტრირება.

```
//      ვირტუალურ საბაზო კლასთან მუშაობის დემონსტრირება
//      საბაზო კლასის გამოცხადება
class Sabazo {
public :
        int cvladi;
        int Naxva() { return cvladi; }
};
//      პირველი მემკვიდრე კლასის გამოცხადება
class Memkvidre1 : public virtual Sabazo {
```

```
public :
        int cvladi1;
        int Naxva1() { return cvladi1 + cvladi; }
};
//      მეორე მემკვიდრე კლასის გამოცხადება
class Memkvidre2 : public virtual Sabazo {
public :
        int cvladi2;
        int Naxva2() { return cvladi2 + cvladi; }
};
//      მემკვიდრე კლასის გამოცხადება, რომელსაც ორი წინაპარი კლასი აქვს
class Memkvidre3 : public Memkvidre1, public Memkvidre2 {
public :
        int cvladi3;
        int Naxva3() { return cvladi3 + cvladi2 + cvladi1 + cvladi; }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Memkvidre3 obj;
obj.cvladi = Convert::ToInt32(textBox1->Text);
obj.cvladi1 = Convert::ToInt32(textBox2->Text);
obj.cvladi2 = Convert::ToInt32(textBox3->Text);
obj.cvladi3 = Convert::ToInt32(textBox4->Text);
label1->Text = obj.Naxva().ToString();
label2->Text = obj.Naxva1().ToString();
label3->Text = obj.Naxva2().ToString();
label4->Text = obj.Naxva3().ToString();
}
```

Memkvidre1 და Memkvidre2 კლასების გამოცხადებისას საბაზო კლასის სახელის წინ რომ არ მიგვეთითებინა virtual სიტყვა, მაშინ

```
return cvladi3 + cvladi2 + cvladi1 + cvladi;
```

ოპერტორის შესრულებისას აღიძვრებოდა შეცდომა. შეცდომას აღიძვრებოდა აგრეთვე

```
obj.cvladi = Convert::ToInt32(textBox1->Text);
```

მინიჭების ოპერატორის შესრულების დროსაც.

მიუხედავად იმისა, რომ საბაზო კლასის სახელის წინ ვუთითებთ virtual სიტყვას, მემკვიდრე კლასი მაინც შეიცავს საბაზო კლასის ასლს. განსხვავება ჩვეულებრივ და ვირტუალურ კლასებს შორის ვლინდება მაშინ, როდესაც საბაზო კლასი მემკვიდრეობით გადაიცემა ერთზე მეტად. ვირტუალური საბაზო კლასის გამოყენების შემთხვევაში მემკვიდრეობითობით მიღებული თითოეული ობიექტი შეიცავს საბაზო კლასის მხოლოდ ერთ ასლს.

ადრეული და გვიანი დაკავშირება

ადრეული დაკავშირება (early binding) ეხება იმ მოვლენებს, რომელთა შესახებ შეგვიძლია გავიგოთ პროგრამის კომპილირების პროცესში. ეს განსაკუთრებით ეხება ფუნქციების გამოძახებებს, რომელთა გაწყობა კომპილირებისას ხდება. ადრეული

დაკავშირების ფუნქციებია: ჩვეულებრივი ფუნქციები, გადატვირთვადი ფუნქციები, არავირტუალური ფუნქცია-წევრები და მეგობრული ფუნქციები. ამ ტიპის ფუნქციების კომპილირებისას ცნობილია ყველა ის ინფორმაცია, რომელიც აუცილებელია მათი გამოძახებისათვის. ადრეული დაკავშირების ძირითადი უპირატერსობაა ის, რომ ის უზრუნველყოფს პროგრამების შესრულების მაღალ სისწრაფეს, რადგანაც კომპილირებისას განისაზღვრება გამოსაძახებელი ფუნქციის საჭირო ვერსია. ადრეული დაკავშირების მთავარი ნაკლია მისი სიხისტე.

გვიანი დაკავშირება (late binding) სრულდება პროგრამის შესრულებისას, რადგან გამოსაძახებელი ფუნქციის მისამართი პროგრამის გაშვებამდე უცნობია. C++ ენაში ვირტუალური ფუნქცია არის გვიანი დაკავშირების ობიექტი. თუ ვირტუალურ ფუნქციასთან მიმართვა სრულდება საბაზო კლასზე მიმთითებლის მეშვეობით, მაშინ მუშაობის პროცესში პროგრამამ უნდა განსაზღვროს თუ რა ტიპის ობიექტს მიმართავს და შემდეგ აირჩიოს თუ რომელი ვირტუალური ფუნქცია უნდა შეასრულოს. გვიანი დაკავშირების მთავარი ღირსებაა მოქნილობა პროგრამის შესრულებისას, ხოლო ნაკლია დაბალი სისწრაფე, რადგან პროგრამის შესრულებისას ხდება ვირტუალური ფუნქციის აჩევა.

თუ რომელი ტიპის დაკავშირებას გამოვიყენებთ, ეს დამოკიდებულია გადასაწყვეტ ამოცანასა და მის მიმართ წაყენებულ მოთხოვნებზე.

object კლასი

C++ ენაში განსაზღვრულია სპეციალური Object კლასი, რომელიც არის საბაზო ყველა სხვა კლასისათვის. სხვა სიტყვებით რომ ვთქვათ, ყველა კლასი არის Object კლასის მემკვიდრე, ე.ი. Object ტიპის მქონე მიმთითებელი შეიძლება მიმართავდეს ნებისმიერი სხვა ტიპის ობიექტს. Object ტიპის ობიექტს შეგვიძლია ნებისმიერი ტიპის მნიშვნელობა მივანიჭოთ. მაგალითი:

```
//      Object ტიპის ობიექტს სხვადასხვა ტიპის მნიშვნელობა ენიჭება
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
System::Object^ obj1;
//      obj1 ობიექტს ენიჭება მთელი რიცხვი
obj1 = Convert::ToInt32(textBox1->Text);
label1->Text = obj1->ToString();
//      obj1 ობიექტს ენიჭება წილადი
obj1 = Convert::ToDouble(textBox2->Text);
label2->Text = obj1->ToString();
//      obj1 ობიექტს ენიჭება ლოგიკური მნიშვნელობა
obj1 = Convert::ToBoolean(textBox3->Text);
label3->Text = obj1->ToString();
//      obj1 ობიექტს ენიჭება სიმბოლო
obj1 = Convert::ToChar(textBox4->Text);
label4->Text = obj1->ToString();
//      obj1 ობიექტს ენიჭება სტრიქონი
obj1 = textBox5->Text;
label5->Text = obj1->ToString();
}
```

C++/CLI კლასები და ჩვენ მიერ განსაზღვრული კლასები (როგორც მნიშვნელობითი, ისე მიმართვითი) მემკვიდრეობით არიან მიღებული System::Object სტანდარტული საბაზო

კლასიდან და შესაბამისად, აქვთ ამ კლასის ყველა შესაძლებლობა.

System::Object საბაზო კლასი პასუხს აგებს ფუნდამენტური ტიპების მნიშვნელობების შეფუთვასა და განფუთვაზე. ეს იმას ნიშნავს, რომ ფუნდამენტური ტიპების მნიშვნელობები საჭიროების შემთხვევაში შეიძლება მოიქცნენ როგორც ობიექტები ან როგორ ჩვეულებრივი ტიპის მნიშვნელობები, მაგალითად, მონაწილეობა მიიღონ არითმეტიკის ოპერაციებში. ფუნდამენტური ტიპის მნიშვნელობები მხოლოდ აუცილებლობის შემთხვევაში გარდაიქმნება ობიექტად, რომელსაც შეგვიძლია მივმართოთ System::Object^ დესკრიპტორის საშუალებით.

მართალია, System::Object არის საბაზო კლასი მნიშვნელობითი კლასებისათვის, ჩვენ არ შეგვიძლია განვსაზღვროთ მნიშვნელობითი კლასები როგორც მემკვიდრეობით მიღებული ამ საბაზო კლასიდან. ჩვენ შეგვიძლია მემკვიდრეობით მივიღოთ მიმართვითი კლასი საბაზო მიმართვითი კლასიდან ზუსტად ისე, როგორც ეს ხდება მშობლიურ C++ენაში.

თავი 9. თვისება, ინტერფეისი, დელეგატი, მოვლენა და სახელების სივრცე

თვისება

თვისება (property) არის მნიშვნელობითი ან მიმართვითი კლასის წევრი. ის საშუალებას გვაძლევს მივიღოთ ან შევცვალოთ ცვლადის (ველის) მნიშვნელობები. თვისება ცვლადისგან იმით განსხვავდება, რომ ცვლადის სახელი მიმართავს მეხსიერების უბანს, რომელშიც მონაცემია მოთავსებული, თვისების სახელი კი - ფუნქციას იძახებს. თვისების გამოყენება მოხერხებულია მაშინ, როდესაც საჭიროა ცვლადის მნიშვნელობებზე შესასრულებელი ოპერაციების კონტროლი. მაგალითად, ცვლადს შეგვიძლია მივანიჭოთ მხოლოდ დადებითი მნიშვნელობები ან მნიშვნელობები გარკვეული დიაპაზონიდან და ა.შ. ბუნებრივია, ამ მიზნის მისაღწევად შეგვიძლია გამოვიყენოთ პრივატული ცვლადი და ფუნქცია, რომელიც მასთან მუშაობს. მაგრამ, გაცილებით მარტივი და უკეთესია პრივატული ცვლადებისა და თვისებების ერთობლივი გამოყენება.

თვისებას აქვს მიმართვის set() და get() ფუნქციები. მიმართვის get ფუნქცია გამოიყენება ცვლადის მნიშვნელობის მისაღებად, set ფუნქცია კი ცვლადისთვის მნიშვნელობის მისანიჭებლად. set ფუნქციას გადაეცემა პარამეტრი, რომელიც შეიცავს ცვლადისათვის მისანიჭებელ მნიშვნელობას. თვისების ძირითადი ღირსებაა ის, რომ მისი სახელი შეიძლება გამოვიყენოთ გამოსახულებებში და მინიჭების ოპერაციებში, როგორც ჩვეულებრივი ცვლადი. ამ დროს ავტომატურად გამოიძახება მიმართვის get და set ფუნქციები. გარდა ამისა, თვისება მართავს ცვლადთან მიმართვას. თვისებაში არ ხდება ცვლადის გამოცხადება. საჭიროების მიხედვით თვისებაში შეიძლება განისაზღვროს მხოლოდ get ან set ფუნქცია, ან ორივე ერთად.

თვისების სინტაქსია:

მიმართვის_მოდიფიკატორი **: property** ***ტიპი თვისების_სახელი***
{
get()
{
get ***ფუნქციის კოდი***
}
set()
{
set ***ფუნქციის კოდი***
}
}

აქ ***ტიპი*** თვისების ტიპია (მაგალითად, int). თვისების განსაზღვრის შემდეგ მისი სახელის გამოყენება იწვევს მიმართვის შესაბამისი ფუნქციის გამოძახებას. თუ თვისების სახელი მითითებულია მინიჭების ოპერატორის მარცხნივ, მაშინ გამოიძახება set ფუნქცია; თუ თვისების სახელი მითითებულია მინიჭების ოპერატორის მარჯვნივ, მაშინ გამოიძახება get ფუნქცია.

კლასი შეიძლება შეიცავდეს ორი სახის თვისებას: ***სკალარულ თვისებას*** და ***ინდექსირებულ თვისებას***. სკალარული თვისება არის ერთი მნიშვნელობა, რომელიც მისაწვდომია თვისების სახელის საშუალებით. ინდექსირებული თვისება არის მნიშვნელობების ნაკრები, რომლის ელემენტებს მივმართავთ თვისების სახელის შემდეგ მოთავსებული ინდექსის საშუალებთ.

თუ თვისება ასოცირებულია კონკრეტულ ობიექტთან, მაშინ ის არის ეგზემპლარის

თვისება. მაგალითად, String ობიექტის ან მასივების Length თვისება არის ეგზემპლარის თვისება. თუ თვისება განსაზღვრულია როგორც სტატიკური, მაშინ ის ასოცირებულია მთლიანად კლასთან. შესაბამისად, თვისების მნიშვნელობა საერთოა კლასის ტიპის ყველა ობიექტისთვის.

სკალარული თვისება

როგორც ვიცით, სკალარულ თვისებას ერთი მნიშვნელობა აქვს. სკალარული თვისების get() ფუნქციამ უნდა გასცეს მონაცემი, რომლის ტიპი თვისების ტიპს ემთხვევა, ხოლო set() ფუნქციის პარამეტრის ტიპი უნდა ემთხვეოდეს თვისების ტიპს. მოცემული პროგრამით ხდება სკალარულ თვისებასთან მუშობის დემონსტრირება, რომელიც გასცემს სამკუთხედის პერიმეტრს.

```
//      სკალარულ თვისებასთან მუშობის დემონსტრირება
value class Samkutxedi {
private :
//
int gverdi1;
int gverdi2;
int gverdi3;
int perimetri;
double fartobi;
public :
//      უარგუმენტებო კონსტრუქტორი
Samkutxedi(int par1, int par2, int par3) : gverdi1(par1), gverdi2(par2), gverdi3(par3) { }
Samkutxedi(int par1, int par2) {
gverdi1 = par1;
gverdi2 = par2;
fartobi = ( gverdi1 * gverdi2 ) / 2;
}
double Naxva_Fartobi() {
return fartobi;
}
//      თვისების განსაზღვრა
property int tviseba
{
        int get()
        {
                return gverdi1 + gverdi2 + gverdi3;
        }
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi obj1 = Samkutxedi(cvladi1, cvladi2);
```

```
Samkutxedi obj2 = Samkutxedi(cvladi1, cvladi2, cvladi3);
label1->Text = L"სამკუთხედის ფართობი = " + obj1.Naxva_Fartobi().ToString();
label2->Text = L"სამკუთხედის პერიმეტრი = " + obj2.tviseba.ToString();
}
```

მოცემული პროგრამით განისაზღვრება ChemiTviseba თვისება, რომელიც cvladi ცვლადს მხოლოდ დადებითი მნიშვნელობებს ანიჭებს.

```
//      თვისებასთან მუშაობის დემონსტრირება
int cvladi = 0;
ref class MartiviTviseba {
public :
//      ChemiTviseba თვისების გამოცხადება
property int ChemiTviseba
{
        int get()
        {
        return cvladi;
        }
        void set(int par1)
        {
        //      cvladi ცვლადს ენიჭება par1 მნიშვნელობა თუ ის დადებითია
        if ( par1 >= 0 ) cvladi = par1;
        }
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
//
MartiviTviseba^ obieqti = gcnew MartiviTviseba();
int ricxvi = Convert::ToInt32(textBox1->Text);

label1->Text = obieqti->ChemiTviseba.ToString();
//      ChemiTviseba თვისებას ენიჭება ricxvi ცვლადის დადებითი მნიშვნელობა
obieqti->ChemiTviseba = ricxvi;
label2->Text = obieqti->ChemiTviseba.ToString();
//      obieqti.ChemiTviseba თვისებისთვის უარყოფითი მნიშვნელობის მინიჭების მცდელობა
obieqti->ChemiTviseba = -ricxvi;
label3->Text = obieqti->ChemiTviseba.ToString();
}
```

თვისების get() და set() ფუნქციები შეგვიძლია გამოვაცხადოთ თვისების შიგნით და განვსაზღვროთ კლასის გარეთ. მოცემული პროგრამით ხდება ამის დემონსტრირება.

```
//      თვისების get() და set() ფუნქციები გამოცხადებულია თვისების
//      შიგნით და განსაზღვრულია კლასის გარეთ
int cvladi = 0;
ref class MartiviTviseba {
public :
//      ChemiTviseba თვისების გამოცხადება
property int ChemiTviseba
```

```
{
	int get();
	void set(int par1);
}
};
//	თვისებების განსაზღვრა კლასის გარეთ
int MartiviTviseba::ChemiTviseba::get()
{
	return cvladi;
}
void MartiviTviseba::ChemiTviseba::set(int par1)
{
	//	cvladi ცვლადს ენიჭება value მნიშვნელობა თუ ის დადებითია
	if ( par1 >= 0 ) cvladi = par1;
}
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
MartiviTviseba^ obieqti = gcnew MartiviTviseba();
int ricxvi = Convert::ToInt32(textBox1->Text);

label1->Text = obieqti->ChemiTviseba.ToString();
//	ChemiTviseba თვისებას ენიჭება ricxvi ცვლადის დადებითი მნიშვნელობა
obieqti->ChemiTviseba = ricxvi;
label2->Text = obieqti->ChemiTviseba.ToString();
//	obieqti.ChemiTviseba თვისებისთვის უარყოფითი მნიშვნელობის მინიჭების მცდელობა
obieqti->ChemiTviseba = -ricxvi;
label3->Text = obieqti->ChemiTviseba.ToString();
}
```

	თვისება შეგვიძლია განვსაზღვროთ მიმართვითი კლასისთვისაც. მოცემული პროგრამით ხდება ამის დემონსტრირება.

```
//	თვისება განსაზღვრულია მიმართვითი კლასისთვის
int cvladi = 0;
ref class MartiviTviseba {
public :
//	ChemiTviseba თვისების გამოცხადება
property int ChemiTviseba
{
	int get()
	{
	return cvladi;
	}
	void set(int par1)
	{
	//	cvladi ცვლადს ენიჭება value მნიშვნელობა თუ ის დადებითია
	if ( par1 >= 0 ) cvladi = par1;
	}
}
```

```
};
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
//
MartiviTviseba^ obieqti = gcnew MartiviTviseba;
int ricxvi = Convert::ToInt32(textBox1->Text);

label1->Text = obieqti->ChemiTviseba.ToString();
//      ChemiTviseba თვისებას ენიჭება ricxvi ცვლადის დადებითი მნიშვნელობა
obieqti->ChemiTviseba = ricxvi;
label2->Text = obieqti->ChemiTviseba.ToString();
//      obieqti.ChemiTviseba თვისებისთვის უარყოფითი მნიშვნელობის მინიჭების მცდელობა
obieqti->ChemiTviseba = -ricxvi;
label3->Text = obieqti->ChemiTviseba.ToString();
}
```

ტრივიალური სკალარული თვისება

ტრივიალური ეწოდება ისეთ სკალარულ თვისებას, რომელიც get() და set() ფუნქციებს არ შეიცავს. ეს ფუნქციები ავტომატურად გამოიყენება ტრივიალური სკალარული თვისების მიმართ. მაგალითი:

```
//      ტრივიალურ სკალარულ თვისებასთან      მუშაობის დემონსტრირება
ref class ChemiTviseba {
public :
        property int tviseba;    //        ტრივიალური თვისება
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);
ChemiTviseba^ obieqti = gcnew ChemiTviseba();
obieqti->tviseba = ricxvi;
label1->Text = obieqti->tviseba.ToString();
}
```

obieqti->tviseba = ricxvi; სტრიქონში ხდება თვისებისათვის მნიშვნელობის მინიჭება, მომდევნო სტრიქონში კი - მნიშვნელობის გაცემა.

ინდექსირებული თვისება

ინდექსირებული თვისება არის თვისების მნიშვნელობა კლასში, რომელთანაც მიმართვა ინდექსის საშუალებით ხორციელდება. სახელის არმქონე თვისებას, რომელთანაც მიმართვა ინდექსის საშუალებით ხორციელდება ***ნაგულისხმევი ინდექსირებული თვისება*** ეწოდება. სახელის მქონე ინდექსირებულ თვისებას ***სახელდებული ინდექსირებული თვისება*** ეწოდება. მოცემული პროგრამით ხდება ნაგულისხმევი ინდექსირებული თვისების გამოყენების დემონსტრირება.

```
//      ნაგულისხმევი ინდექსირებული თვისების გამოყენების დემონსტრირება
ref class IndexTviseba {
private :
        array<System::String^>^ Gvarebi;        //
```

```
public :
	IndexTviseba(...array<System::String^>^ gvarebi) : Gvarebi(gvarebi) { }
	//	სკალარული თვისება ინახავს სახელების რაოდენობას
	property int GvarebisRaodenoba
	{
		int get()
		{
			return Gvarebi->Length;
		}
	}
	//	გაჩუმებითი ინდექსირებადი თვისება გასცემს გვარს
	property System::String^ default[int]
	{
		System::String^ get(int index)
		{
			if ( index >= Gvarebi->Length ) return L"არასწორი ინდექსი";
			return Gvarebi[index];
		}
		void set(int index, System::String^ gvari)
		{
			if ( index >= Gvarebi->Length )
			System::Windows::Forms::MessageBox::Show( L"არასწორი ინდექსი");
			Gvarebi[index] = gvari;
		}
	}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);
IndexTviseba^ obieqti = gcnew IndexTviseba(L"სამხარაძე", L"კაპანაძე", L"ჭუმბურიძე",
                                           L"კირვალიძე", L"ხუციშვილი");
for ( int ind = 0; ind < obieqti->GvarebisRaodenoba; ind++ )
	label1->Text += obieqti[ind]->ToString() + L"\n";
obieqti[obieqti->GvarebisRaodenoba-3] = L"შენგელია";
for ( int ind = 0; ind < obieqti->GvarebisRaodenoba; ind++ )
	label2->Text += obieqti[ind]->ToString() + L"\n";
}
```

პროგრამაში გამოიყენება ნაგულისხმევი ინდექსირებული თვისება, რადგან მისი სახელი განსაზღვრულია default საკვანძო სიტყვით. ასეთ თვისებასთან მიმართვისათვის ობიექტის სახელის შემდეგ, კვადრატულ ფრჩხილებში, უნდა მივუთითოთ ინდექსი. ინდექსს შეიძლება ნებისმიერი ტიპი ჰქონდეს. ინდექსირებულ თვისებას შეიძლება ჰქონდეს ინდექსის რამდენიმე პარამეტრი.

ინდექსირებადი თვისებისთვის, რომელთანაც მიმართვა ერთი ინდექსით ხორციელდება, get() ფუნქციას უნდა ჰქონდეს პარამეტრი, რომელიც განსაზღვრავს იმავე ტიპის ინდექსს, რა ტიპიც მითითებულია თვისების განსაზღვრისას. ასეთი ინდექსირებადი თვისებისთვის set() ფუნქციას უნდა ჰქონდეს ორი პარამეტრი: პირველი პარამეტრია ინდექსი, მეორე კი - ახალი მნიშვნელობა, რომელიც ინდექსით განსაზღვრულ თვისებას უნდა მიენიჭოს.

თუ თვისების განსაზღვრაში default საკვანძო სიტყვას სახელით შევცვლით, მაშინ მივიღებთ სახელდებულ ინდექსირებულ თვისებას. მოცემულ პროგრამაში ხდება მასთან მუშაობის დემონსტრირება.

```
//      სახელდებულ ინდექსირებულ თვისებასთან მუშაობის დემონსტრირება
ref class IndexTviseba_1 {
private :
        array<System::String^>^ Gvarebi;
public :
        IndexTviseba_1(...array<System::String^>^ gvarebi) : Gvarebi(gvarebi) { }
        //      სკალარული თვისება ინახავს სახელების რაოდენობას
        property int GvarebisRaodenoba
        {
                int get()
                {
                        return Gvarebi->Length;
                }
        }
        //      სახელდებული ინდექსირებული თვისება გასცემს და ცვლის გვარს
        property System::String^ Tviseba[int]
        {
                System::String^ get(int index)
                {
                        if ( index >= Gvarebi->Length ) return L"არასწორი ინდექსი";
                        return Gvarebi[index];
                }
                void set(int index, System::String^ gvari)
                {
                        if ( index >= Gvarebi->Length )
                            System::Windows::Forms::MessageBox::Show( L"არასწორი ინდექსი");
                        Gvarebi[index] = gvari;
                }
        }
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi = Convert::ToInt32(textBox1->Text);
IndexTviseba_1^ obieqti = gcnew
        IndexTviseba_1(L"სამხარაძე", L"კაპანაძე", L"ჭუმბურიძე", L"კირვალიძე", L"ხუციშვილი");

for ( int ind = 0; ind < obieqti->GvarebisRaodenoba; ind++ )
        label1->Text += obieqti->Tviseba[ind]->ToString() + L"\n";
obieqti->Tviseba[obieqti->GvarebisRaodenoba-3] = L"შენგელია";
for ( int ind = 0; ind < obieqti->GvarebisRaodenoba; ind++ )
        label2->Text += obieqti->Tviseba[ind]->ToString() + L"\n";
}
```

სტატიკური თვისება

კლასის სტატიკური წევრის მსგავსად კლასის სტატიკური თვისება განისაზღვრება მთელი კლასისთვის და არსებობს ერთ ეგზემპლარად ამ კლასის ყველა ობიექტისთვის. კლასის სტატიკური თვისებები არსებობენ ამ კლასის ტიპის ობიექტების არსებობისაგან დამოუკიდებლად. მოცემული პროგრამით ხდება სტატიკურ თვისებასთან მუშაობის დემონსტრირება.

```
//      სტატიკურ თვისებასთან მუშაობის დემონსტრირება
ref class Samkutxedi_1 {
private :
int gverdi1;
int gverdi2;
int gverdi3;
public :
static int perimetri;
//      უარგუმენტებო კონსტრუქტორი
Samkutxedi_1(int par1, int par2, int par3) {
gverdi1 = par1;
gverdi2 = par2;
gverdi3 = par3;
perimetri = gverdi1 + gverdi2 + gverdi3;
}
static property int tviseba
{
        int get()
        {
                return perimetri;
        }
}
};
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
int cvladi3 = Convert::ToInt32(textBox3->Text);
 Samkutxedi_1^ obj1 = gcnew Samkutxedi_1(cvladi1, cvladi2, cvladi3);
label1->Text = Samkutxedi_1::tviseba.ToString();
}
```

ინტერფეისული კლასი

ინტერფეისი არის კლასი, რომელიც განსაზღვრავს ფუნქციების ნაკრებს, რომლებიც რეალიზებული უნდა იყოს სხვა კლასების მიერ. ინტერფეისული კლასის (ინტერფეისის) რეალიზება შეუძლია როგორც მნიშვნელობით, ისე მიმართვით კლასებს. ინტერფეისული კლასის გამოცხადება ხდება interface class ან interface struct საკვანძო სიტყვების გამოყენებით.

ინტერფეისულ კლასში შეგვიძლია გამოვაცხადოთ ფუნქციები, თვისებები, სტატიკური ველები, მოვლენები და სტატიკური კონსტრუქტორები. ინტერფეისში ხდება ფუნქციების მხოლოდ გამოცხადება. მათი განსაზღვრა უნდა მოხდეს იმ კლასში, რომელიც ახდენს მოცემული ინტერფეისის რეალიზებას. ინტერფეისული კლასი შეიძლება შეიცავდეს ნებისმიერი სახის ჩადგმული კლასების განსაზღვრას. ერთი ინტერფეისი შეიძლება იყოს მეორე ინტერფეისის მემკვიდრე. ნაგულისხმევად, ინტერფეისული კლასის ყველა წევრი არის ღია (public). არსებობს შეთანხმება, რომ ინტერფეისული კლასის სახელი I ასოთი იწყებოდეს (თუმცა ამ შეთანხმების დაცვა თქვენზეა დამოკიდებული). მოცემულ პროგრამაში Samkutxedi კლასი არის Sabazo საბაზო კლასისა და ISamkutxedi და INaxva ინტერფეისული კლასების მემკვიდრე.

```
//      ინტერფეისთან მუშაობის დემონსტრირება
//      ინტერფეისული კლასის გამოცხადება
interface class ISamkutxedi {
        int Perimetri(int par1, int par2, int par3);
        double Fartobi();
};
//      ინტერფეისული კლასის გამოცხადება
interface class INaxva {
        void Naxva();
};
//      საბაზო კლასის გამოცხადება
ref class Sabazo {
public :
        int gverdi1;
        int gverdi2;
        int gverdi3;
};
//      მემკვიდრე კლასის გამოცხადება
ref class Samkutxedi : Sabazo, ISamkutxedi, INaxva {
private :
        int perimetri;
public :
        Samkutxedi(int par1, int par2, int par3) {
                gverdi1 = par1;
                gverdi2 = par2;
                gverdi3 = par3;
        }
        virtual int Perimetri(int par1, int par2, int par3) {
                gverdi1 = par1;
                gverdi2 = par2;
                gverdi3 = par3;
                perimetri = gverdi1 + gverdi2 + gverdi3;
                return perimetri;
        }
        virtual double Fartobi() {
                return ( gverdi1 + gverdi2 ) / 2;
        }
        virtual void Naxva() {
```

```
            System::Windows::Forms::MessageBox::Show(L"პერიმეტრი = " + perimetri.ToString());
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi^ obj1 = gcnew Samkutxedi(ricxvi1, ricxvi2, ricxvi3);
int Perimetri = obj1->Perimetri(ricxvi1, ricxvi2, ricxvi3);
double Fartobi = obj1->Fartobi();
label1->Text = Perimetri.ToString();
label2->Text = Fartobi.ToString();
obj1->Naxva();
}
```

აქ Samkutxedi კლასი მემკვიდრეობით იღებს Sabazo საბაზო კლასისა და ISamkutxedi და INaxva ინტერფეისული კლასების წევრებს. ISamkutxedi და INaxva ინტერფეისულ კლასებში განსაზღვრული ფუნქციები აბსტრაქტულია, რადგანაც ინტერფეისები არასოდეს არ შეიცავენ ფუნქციების განსაზღვრას. აბსტრაქტული კლასის გარდა, ნებისმიერმა კლასმა, რომელიც ახდენს ინტერფეისის რეალიზებას, უნდა მოახდინოს ინტერფეისის ყველა ფუნქციის განსაზღვრა.

ჩვენ შეგვიძლია გამოვიყენოთ ISamkutxedi ტიპის დესკრიპტორი იმისათვის, რომ შევინახოთ იმ კლასის ობიექტის მისამართი, რომელიც ამ ინტერფეისის რეალიზებას ახდენს. ამიტომ, ISamkutxedi ტიპის დესკრიპტორი შეგვიძლია გამოვიყენოთ Samkutxedi^ ტიპის ობიექტთან მიმართვისათვის. მაგალითი:

```
//      ინტერფეისის ტიპის დესკრიპტორი გამოიყენება მიმართვითი კლასის
//      ტიპის ობიექტთან მიმართვისათვის
//      ინტერფეისული კლასის გამოცხადება
interface class ISamkutxedi {
        int Perimetri(int par1, int par2, int par3);
        double Fartobi();
};
//      ინტერფეისული კლასის გამოცხადება
interface class INaxva {
        void Naxva();
};
//      საბაზო კლასის გამოცხადება
ref class Sabazo {
public :
        int gverdi1;
        int gverdi2;
        int gverdi3;
};
//      მემკვიდრე კლასის გამოცხადება
ref class Samkutxedi : Sabazo, ISamkutxedi, INaxva {
private :
        int perimetri;
```

```
public :
	Samkutxedi(int par1, int par2, int par3) {
		gverdi1 = par1;
		gverdi2 = par2;
		gverdi3 = par3;
	}
	virtual int Perimetri(int par1, int par2, int par3) {
		gverdi1 = par1;
		gverdi2 = par2;
		gverdi3 = par3;
		perimetri = gverdi1 + gverdi2 + gverdi3;
		return perimetri;
	}
	virtual double Fartobi() {
		return ( gverdi1 + gverdi2 ) / 2;
	}
	virtual void Naxva() {
		System::Windows::Forms::MessageBox::Show(L"პერიმეტრი = " + perimetri.ToString());
	}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
int ricxvi3 = Convert::ToInt32(textBox3->Text);
Samkutxedi^ obj1 = gcnew Samkutxedi(ricxvi1, ricxvi2, ricxvi3);
ISamkutxedi^ obj2(obj1);
double s1 = obj2->Fartobi();
int s2 = obj2->Perimetri(ricxvi1, ricxvi2, ricxvi3);
label1->Text = s1.ToString();
label2->Text = s2.ToString();
}
```

დელეგატი

დელეგატი ესაა ობიექტი, რომელიც ფუნქციას მიმართავს. დელეგატები გამოიყენება ფუნქციების გამოძახებისათვის. ამასთან, გამოსაძახებელი ფუნქციის არჩევა ხდება დინამიკურად, პროგრამის შესრულებისას. ამიტომ, დელეგატები იმ შემთხვევაში გამოიყენება, როდესაც წინასწარ არ ვიცით თუ რომელი ფუნქცია უნდა გამოვიძახოთ. დელეგატების გამოყენების კიდევ ერთი უპირატესობა იმაში მდგომარეობს, რომ ისინი უზრუნველყოფენ მოვლენებს.

დელეგატი ინახავს გამოსაძახებელი ფუნქციის სახელს, მის მიერ დაბრუნებული შედეგის ტიპსა და პარამეტრების სიას ანუ ფუნქციის სიგნატურას (ხელმოწერას).

როგორც ცნობილია, ფუნქციისათვის მეხსიერებაში გარკვეული ზომის უბანი გამოიყოფა. ამ უბნის მისამართი არის ფუნქციის შესვლის წერტილი. სწორედ ამ მისამართის ინიცირება ხდება ფუნქციის გამოძახებისას. ფუნქციის მისამართი ენიჭება დელეგატს. თუ

დელეგატი მიმართავს ფუნქციას, მაშინ მოცემული ფუნქცია შეგვიძლია გამოვიძახოთ ამ დელეგატის საშუალებით. გარდა ამისა, ერთი და იგივე დელეგატი შეგვიძლია გამოვიყენოთ სხვადასხვა ფუნქციის გამოძახებისათვის. ამისათვის მას უნდა მივანიჭოთ ამ ფუნქციებზე მიმართვა. ამრიგად, დელეგატი არის „მშობლიური“ C++ ენის ფუნქციაზე მიმთითებლის მსგავსი საშუალება.

დელეგატის გამოცხადების სინტაქსია:

მიმართვის_მოდიფიკატორი **delegate** *ტიპი დელეგატის_სახელი(პარამეტრების_სია)*;

აქ *ტიპი* არის იმ მნიშვნელობის ტიპი, რომელსაც დელეგატის მიერ გამოძახებული ფუნქცია აბრუნებს.

დელეგატზე მიმართვის ტიპს საბაზო კლასად აქვს System::Delegate. ამიტომ, დელეგატის ტიპი ყოველთვის მემკვიდრეობით იღებს ამ კლასის წევრებს. დელეგატის გამოცხადების მაგალითია:

```
public delegate void Delegati(int cvladi);
```

აქ განისაზღვრება Delegati დელეგატზე მიმართვის ტიპი. Delegati დელეგატი არის მიღებული System::Delegate კლასიდან. Delegati ტიპის ობიექტი შეიძლება შეიცავდეს ერთ ან მეტ ფუნქციაზე მიმთითებელს. ფუნქციებს უნდა ჰქონდეს void დაბრუნების ტიპი და int ტიპის ერთი პარამეტრი. ფუნქციები, რომლებზეც დელეგატი მიუთითებს, შეიძლება იყოს ეგზემპლარის ფუნქციები ან სტატიკური ფუნქციები.

დელეგატის კონსტრუქტორს შეიძლება ჰქონდეს ერთი ან ორი არგუმენტი. ერთი არგუმენტის შემთხვევაში, დელეგატის კონსტრუქტორის არგუმენტი უნდა იყოს კლასის სტატიკური ფუნქცია-წევრი ან გლობალური ფუნქცია, რომლის პარამეტრების სია და დაბრუნების ტიპი ემთხვევა დელეგატის გამოცხადებაში მითითებულ პარამეტრების სიასა და დაბრუნების ტიპს. მაგალითი:

```
Delegati^ delegati1 = gcnew Delegati(Klasi1::Funqcia1);
```

ორი არგუმენტის შემთხვევაში, პირველ არგუმენტს უნდა გადავცეთ CLR გროვაში მოთავსებული ობიექტზე მიმართვა, მეორე არგუმენტს კი - ამ ობიექტის ტიპის ეგზემპლარის ფუნქციის მისამართი. მაგალითი:

```
Klasi1^ obj = gcnew Klasi1;
Delegati^ delegati2 = gcnew Delegati(obj, & Klasi1::Funqcia3);
```

მოცემული პროგრამით ხდება დელეგატთან მუშაობის დემონსტრირება.

```
//      დელეგატთან მუშაობის დემონსტრირება
//      დელეგატის გამოცხადება
public delegate void Delegati(int cvladi);
//
public ref class Klasi1 {
public :
static void Funqcia1(int par) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia1 - პარამეტრი = " + par.ToString());
}
static void Funqcia2(int par) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia2 - პარამეტრი = " + par.ToString());
}
void Funqcia3(int par) {
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia3 - პარამეტრი = " + par.ToString());
}
Klasi1() : cvladi(1) { }
Klasi1(int par) : cvladi(par) { }
```

```
protected :
        int cvladi;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Delegati^ delegati1 = gcnew Delegati(Klasi1::Funqcia1);
delegati1->Invoke(50);          //      გამოიძახება Funqcia1 ფუნქცია
delegati1(35);                  //      გამოიძახება Funqcia1 ფუნქცია
//      delegati1 დელეგატს ემატება Funqcia2 ფუნქციის მისამართი
delegati1 += gcnew Delegati(Klasi1::Funqcia2);
delegati1(100);                 //      გამოიძახება Funqcia1 და Funqcia2 ფუნქციები ფუნქციები
//      delegati1 დელეგატიდან იშლება Funqcia1 ფუნქციის მისამართი
delegati1 -= gcnew Delegati(Klasi1::Funqcia1);
delegati1(200);                 //      გამოიძახება Funqcia2 ფუნქცია
//
Klasi1^ obj = gcnew Klasi1;
Delegati^ delegati2 = gcnew Delegati(obj, & Klasi1::Funqcia3);
delegati2(77);                  //      გამოიძახება Funqcia3 ფუნქცია
//      delegati1 დელეგატს ემატება delegati2 დელეგატის გამოძახებების სია
delegati1 += delegati2;
delegati1(15);                  //      გამოიძახება Funqcia2 და Funqcia3 ფუნქციები ფუნქციები
}
```

Klasi1 კლასში განსაზღვრულია სამი ფუნქცია, რომელთაგან ორი სტატიკურია (Funqcia1 და Funqcia2) და ერთი - ეგზემპლარის ფუნქცია (Funqcia3). Delegati ტიპის დელეგატი შეგვიძლია ასე შევქმნათ:

```
Delegati^ delegati1 = gcnew Delegati(Klasi1::Funqcia1);
```

delegati1 ობიექტი შეიცავს Klasi1 კლასის Funqcia1 ფუნქციის მისამართს. ამიტომ, delegati1 დელეგატის გამოძახებისას გამოძახებული იქნება Funqcia1 ფუნქცია. ამ ფუნქციას გადაეცემა ის არგუმენტები, რომლებიც მითითებული იყო დელეგატის გამოძახებისას. დელეგატის გამოძახება ორი გზით შეგვიძლია:

```
delegati1->Invoke(50);
```

ან

```
delegati1(35);
```

ამ დროს შესრულდება ყველა იმ ფუნქციის გამოძახება, რომელთა მისამართები მოთავსებულია delegati1 დელეგატის გამოძახებების სიაში. ჩვენ შემთხვევაში, გამოძახებების სიაში ერთი ფუნქციის (Funqcia1) მისამართია. გამოძახებების სიაში შეგვიძლია სხვა ფუნქციის მისამართის დამატება + ოპერატორის გამოყენებით. delegati1 დელეგატს დავუმატოთ Funqcia2 ფუნქციის მისამართი:

```
delegati1 += gcnew Delegati(Klasi1::Funqcia2);
```

ახლა delegati1 დელეგატის გამოძახებების სიაში Funqcia1 და Funqcia2 ფუნქციების მისამართებია მოთავსებული. მივიღეთ ახალი delegati1 ობიექტი. delegati1 დელეგატის გამოძახების შემთხვევაში, ჯერ შესრულდება Funqcia1 ფუნქცია, შემდეგ კი - Funqcia2 ანუ ფუნქციები იმ მიმდევრობით გამოიძახება, რა მიმდევრობითაც მოხდა მათი მისამართების დამატება გამოძახებების სიაში.

შეგვიძლია, აგრეთვე წავშალოთ რომელიმე ფუნქციის მისამართი გამოძახებების სიიდან „-“ ოპერატორის გამოყენებით. delegati1 დელეგატის გამოძახებების სიიდან წავშალოთ Funqcia1 ფუნქციის მისამართი:

```
delegati1 -= gcnew Delegati(Klasi1::Funqcia1);
```

ამ შემთხვევაში გამომახებების სიაში დარჩება Funqcia2 ფუნქციის მისამართი. მივიღეთ ახალი delegati1 ობიექტი. ამ დელეგატის გამოძახების შემთხვევაში შესრულდება Funqcia2 ფუნქცია. თუ დელეგატის გამომახებების სიიდან წავშლით ყველა მიმთითებელს, მაშინ სიაში მოთავსებული იქნება nullptr.

ამრიგად, დელეგატის გამოყენებით შეგვიძლია სტატიკური და ჩვეულებრივი ფუნქციების გამოძახებების კომბინირება ერთი დელეგატის გამოძახებების სიაში.

ერთ დელეგატს შეგვიძლია დავუმატოთ მეორე დელეგატის გამოძახებების სია. მაგალითად:

```
delegati1 += delegati2;
```

ამ შემთხვევაში delegati1 დელეგატის გამოძახებების სიას დაემატება delegati2 დელეგატის გამოძახებების სია. შედეგად, delegati1 დელეგატის გამოძახებების სიაში აღმოჩნდება Funqcia1 და Funqcia2 ფუნქციების მისამართები. ამიტომ, delegati1 დელეგატის გამოძახებისას შესრულდება ორივე ეს ფუნქცია.

დელეგატების გამოყენებით შესაძლებელია გადატვირთული ფუნქციის მისამართის მიღება და გამოძახება. მოცემული პროგრამით ხდება ამის დემონსტრირება:

```
//      გადატვირთული ფუნქციების მისამართების მიღება და გამოძახება
//      დელეგატების გამოცხადება
public delegate void Delegati1(int cvladi);
public delegate void Delegati2(int cvladi, System::Char simbolo);
ref class Klasi {
public :
        void Funqcia(int par1) {
        for ( ; par1; par1-- ) System::Windows::Forms::MessageBox::Show(par1.ToString() + "   ");
}
void Funqcia(int par1, System::Char simbolo) {
        for ( ; par1; par1-- ) System::Windows::Forms::MessageBox::Show(simbolo.ToString() + "   ");
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
label1->Text = ""; label2->Text = "";
int cvladi1 = Convert::ToInt32(textBox1->Text);
Klasi^ obj = gcnew Klasi();

Delegati1^ delegati1 = gcnew Delegati1(obj, &Klasi::Funqcia);
Delegati2^ delegati2 = gcnew Delegati2(obj, &Klasi::Funqcia);

delegati1(cvladi1);
delegati2(cvladi1, L'ლ');
}
```

მოვლენა

მოვლენა არის ავტომატური უწყება რაიმე მომხდარ მოქმედებაზე. მოვლენის საშუალებით ერთი ობიექტი მეორეს ატყობინებს, რომ რაღაც მოხდა. მოვლენის მაგალითებია:

კლავიატურის კლავიშის დაჭერა, კლავიატურის კლავიშის აშვება, თაგვის მარცხენა კლავიშზე დაჭერა, თაგვის მარჯვენა კლავიშზე დაჭერა, თაგვის კლავიშის აშვება და ა.შ.

ობიექტისთვის, რომელიც მისი განსაზღვრის (კოდის) მიხედვით უნდა რეაგირებდეს რაიმე მოვლენაზე, რეგისტრირდება ამ ***მოვლენის დამამუშავებელი*** (ფუნქცია). მოვლენების დამამუშავებლები იქმნება დელეგატების საფუძველზე.

მოვლენა მიმართვითი კლასის წევრია და მისი გამოცხადება ხდება event საკვანძო სიტყვის გამოყენებით. მოვლენის გამოცხადების სინტაქსია:

event *დელეგატის_სახელი^ მოვლენის_სახელი;*

აქ ***დელეგატის_სახელი*** იმ დელეგატის სახელია, რომელიც გამოიყენება მოვლენის დასამუშავებლად, ხოლო ***მოვლენის_სახელი*** არის შესაქმნელი მოვლენის (მოვლენა-ობიექტის) სახელი.

მოცემული პროგრამით ხდება მოვლენასთან მუშაობის დემონსტრირება.

```
//      მოვლენასთან მუშაობის დემონსტრირება
//      დელეგატის გამოცხადება
public delegate void Delegati(System::String^ striqoni);
//
public ref class Klasi1 {
public :
//      Movlena მოვლენის გამოცხადება
event Delegati^ Movlena;
//      მოვლენის აღმძვრელი ფუნქცია
 void Funqcia_Movlena()
 {
Movlena(L"ანა");
Movlena(L"საბა");
 }
};
//      კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებლებს
public ref class Klasi_Damamushavebeli {
public :
void Funqcia_Damamushavebeli_1(System::String^ par)
{
System::Windows::Forms::MessageBox::Show
                                (L"სრულდება Funqcia_Damamushavebeli_1 ფუნქცია - " + par);
}
void Funqcia_Damamushavebeli_2(System::String^ par)
{
System::Windows::Forms::MessageBox::Show
                                (L"სრულდება Funqcia_Damamushavebeli_2 ფუნქცია - " + par);
}
};
//
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi1^ obj = gcnew Klasi1;
Klasi_Damamushavebeli^ dam1 = gcnew Klasi_Damamushavebeli;
//      Klasi1 კლასის წევრი Movlena მოვლენის დამამუშავებლის დამატება
obj->Movlena += gcnew Delegati(dam1, &Klasi_Damamushavebeli::Funqcia_Damamushavebeli_1);
```

```
obj->Funqcia_Movlena();        //        Movlena მოვლენის ინიცირება
//        Movlena მოვლენის დამუშავების წესის შეცვლა
obj->Movlena -= gcnew Delegati(dam1, &Klasi_Damamushavebeli::Funqcia_Damamushavebeli_1);
obj->Movlena += gcnew Delegati(dam1, &Klasi_Damamushavebeli::Funqcia_Damamushavebeli_2);
obj->Funqcia_Movlena();
}
```

Klasi1 კლასში განსაზღვრულია Movlena მოვლენა. ჩვენ შემთხვევაში, Movlena მოვლენა არის Klasi1 კლასის ეგზემპლარის წევრი, თუმცა შეგვიძლია განვსაზღვროთ სტატიკური მოვლენებიც. მოვლენა შეგვიძლია გამოვაცხადოთ, აგრეთვე, როგორ სტატიკური (static), ისე ვირტუალური (virtual). Movlena მოვლენის ინიცირებისას შესრულდება Delegati დელეგატით განსაზღვრული ფუნქციები. Klasi1 კლასში განსაზღვრულია Funqcia_Movlena() ფუნქცია, რომელიც ახდენს ორი Movlena მოვლენის ინიცირებას. თითოეულ მათგანს თავისი არგუმენტები აქვს. ეს არგუმენტები იმ ფუნქციებს გადაეცემა, რომლებიც დარეგისტრირებულია Movlena მოვლენის შესახებ უწყების მისაღებად. პრაქტიკულად, მოვლენის ინიცირება არსით იგივეა, რაც დელეგატის გამოძახება.

Klasi_Damamushavebeli კლასში განსაზღვრულია Movlena მოვლენის დამამუშავებელი ორი ფუნქცია. მოვლენის დამამუშავებელი ფუნქციების დარეგისტრირებამდე ჯერ უნდა შევქმნათ Klasi1 ობიექტი:

```
Klasi1^ obj = gcnew Klasi1;
```

შემდეგ კი - დავარეგისტრიროთ საჭირო ფუნქციები:

```
Klasi_Damamushavebeli^ dam1 = gcnew Klasi_Damamushavebeli;
obj->Movlena += gcnew Delegati(dam1, &Klasi_Damamushavebeli::Funqcia_Damamushavebeli_1);
```

აქ ჯერ იქმნება Klasi_Damamushavebeli ტიპის dam1 ობიექტი, შემდეგ კი obj ობიექტის Movlena წევრს ვუმატებთ Delegati ტიპის დელეგატის ეგზემპლარს. ეს პროცესი დელეგატისთვის ფუნქციებზე მიმთითებლების დამატების პროცესის ანალოგიურია.

შემდეგ Delegati დელეგატი გამოძახებების სიიდან იშლება Funqcia_Damamushavebeli_1 ფუნქციაზე მიმთითებელი და ემატება Funqcia_Damamushavebeli_2 ფუნქციაზე მიმთითებელი. შედეგად, მოვლენის აღძვრის შემთხვევაში მოვლენას დაამუშავებს Funqcia_Damamushavebeli_2 ფუნქცია.

მოცემულ პროგრამაში მოვლენა აღიძვრება მაშინ, როდესაც შეტანილია ლუწი რიცხვი.

```
//        მოვლენა აღიძვრება მაშინ, როდესაც შეტანილია ლუწი რიცხვი
//        დელეგატის გამოცხადება
public delegate void Delegati2(int ricxvi);
//        მოვლენის შემცველი კლასი
public ref class Klasi {
public :
//        მოვლენა იძახებს Delegati დელეგატთან ასოცირებულ ფუნქციებს
event Delegati2^ Movlena;
//        მოვლენის აღმძვრელი ფუნქცია
 void Funqcia_Movlena(int par1)
 {
        if ( par1 % 2 == 0 ) Movlena(par1);
 }
};
//        კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებელს
public ref class Klasi_Damamushavebeli2 {
public :
```

```
void Funqcia_Damamushavebeli_1(int par2)
{
System::Windows::Forms::MessageBox::Show(L" აღიძვრა მოვლენა\nრიცხვი " +
                                                    par2.ToString() + L" არის ლუწი");
}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
Klasi^ obj2 = gcnew Klasi;
Klasi_Damamushavebeli2^ dam2 = gcnew Klasi_Damamushavebeli2;
//
obj2->Movlena += gcnew Delegati2(dam2, &Klasi_Damamushavebeli2::Funqcia_Damamushavebeli_1);
obj2->Funqcia_Movlena(cvladi);          //          Movlena მოვლენის ინიცირება
}
```

სახელების სივრცე

სახელების სივრცე თავიდან გვაცილებს პროგრამის კლასების სახელებს შორის კონფლიქტს. დავუშვათ, რომ ჩვენი პროგრამა იყენებს სხვა პროგრამისტის მიერ შემუშავებულ კლასს, რომლის სახელია Manqana. დავუშვათ ასევე, რომ ჩვენც დაგვჭირდა ამავე სახელის კლასის განსაზღვრა. ასეთ შემთხვევაში მოგვიწევს სხვა სახელის გამოყენება. სწორედ ასეთი პრობლემების გადასაწყვეტად გამოიყენება სახელების სივრცე.

სახელების სივრცე განსაზღვრავს გამოცხადების უბანს, რომელიც საშუალებას გვაძლევს სახელების თითოეული ნაკრები შევინახოთ სხვა ნაკრებისაგან ცალკე. სახელების ერთ სივრცეში გამოცხადებული სახელები არ არიან კონფლიქტში სახელების სხვა სივრცეში გამოცხადებულ ასეთივე სახელებთან.

System სახელების სივრცეში მრავალი კლასია განსაზღვრული, ამიტომ თითოეული პროგრამა იწყება დირექტივით:

```
using namespace System;
```

სახელების სივრცის გამოცხადება

სახელების სივრცის გამოცხადება ხდება namespace საკვანძო სიტყვის საშუალებით. მისი სინტაქსია:

namespace ***სახელების_სივრცის_სახელი***

{

სახელების სივრცის წევრები

}

სახელების სივრცეში განსაზღვრული ყველა ელემენტი, როგორიცაა კლასები, დელეგატები, ჩამოთვლები, ინტერფეისები ან სხვა სახელების სივრცეები, იმყოფება ამ სახელების სივრცის მხედველობის უბნის საზღვრებში.

ქვემოთ მოცემული პროგრამით ხდება სახელების სივრცესთან მუშაობის დემონსტრირება.

```
//          სახელების სივრცესთან მუშაობა
#pragma once
```

```
namespace Sivrce_1
{
int ricxvi1 = 10;
int ricxvi2;
}

namespace Saxelebis_Sivrce {
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
. . .
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Sivrce_1::ricxvi1 = Convert::ToInt32(textBox1->Text);
Sivrce_1::ricxvi2 = Convert::ToInt32(textBox2->Text);
label1->Text = Sivrce_1::ricxvi1.ToString() + "   " + Sivrce_1::ricxvi2.ToString();
}
```

Sivrce_1 სახელების სივრცე უნდა გამოვაცხადოთ პროგრამის დასაწყისში #pragma once დირექტივის შემდეგ. ამ სივრცეში გამოცხადებულია ორი მთელი რიცხვი. იმისათვის, რომ ამ რიცხვებთან ვიმუშაოთ, პროგრამაში მათი სახელების წინ უნდა მოვათავსოთ Sivrce_1:: პრეფიქსი. :: სიმბოლოების შეტანის შემდეგ გაიხსნება სია, საიდანაც ავირჩევთ საჭირო ცვლადს. სახელების სივრცეში განსაზღვრულ წევრებთან ასეთი სახით მუშაობა საკმაოდ მოუხერხებელია. იმისათვის, რომ სახელების სივრცის სახელი არ მივუთითოთ ყოველთვის მის წევრებთან მიმართვისას, ის უნდა მივუთითოთ using namespace დირექტივების ბლოკში. using namespace დირექტივის სინტაქსია:

using namespace ***სახელების_სივრცის_სახელი***[**::***წევრის_სახელი*]**;**

using namespace დირექტივების ბლოკში using namespace Sivrce_1; დირექტივის ჩართვის შემდეგ ზემოთ მოცემული პროგრამა შემდეგ სახეს მიიღებს:

```
#pragma once

namespace Sivrce_1
{
int ricxvi1 = 10;
int ricxvi2;
}

namespace Saxelebis_Sivrce {
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
```

```
        using namespace Sivrce_1;
. . .
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
label1->Text = ricxvi1.ToString() + "   " + ricxvi2.ToString();
}
```

როგორც ვხედავთ, Sivrce_1 სახელების სივრცეში განსაზღვრულ ცვლადებთან მიმართვისათვის აღარ გამოიყენება Sivrce_1:: პრეფიქსი.

ერთ პროგრამაში შეიძლება ჩავრთოთ რამდენიმე სახელების სივრცე. მაგალითი:

```
//      განსაზღვრულია რამდენიმე სახელების სივრცე
#pragma once

namespace Sivrce_1
{
int ricxvi1 = 10;
int ricxvi2;
}
//
namespace Sivrce_2
{
int ricxvi1 = 55;
int ricxvi3;
}
//
namespace Sivrce_1
{
int ricxvi4 = 50;
int ricxvi5;
}

namespace Saxelebis_Sivrce {
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
        using namespace Sivrce_1;
. . .
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
ricxvi5 = ricxvi1 + ricxvi2 + ricxvi4;
label1->Text = Sivrce_1::ricxvi1.ToString() + "   " + ricxvi2.ToString() + "   " + ricxvi5.ToString();
```

```
//
Sivrce_2::ricxvi1 = Convert::ToInt32(textBox1->Text);
Sivrce_2::ricxvi3 = Convert::ToInt32(textBox2->Text);
label2->Text = Sivrce_2::ricxvi1.ToString() + "   " + Sivrce_2::ricxvi3.ToString();
}
```

აქ გამოცხადებულია სახელების ორი სივრცე: Sivrce_1 და Sivrce_2. ამასთან, ჯერ გამოცხადებულია Sivrce_1, შემდეგ - Sivrce_2. შემდეგ კი ისევ გრძელდება Sivrce_1-ის გამოცხადება. using namespace ბლოკში გამოცხადებულია მხოლოდ Sivrce_1. საქმე ისაა, რომ სახელების ორივე სივრცეში გამოცხადებულია ერთი და იგივე სახელის მქონე ricxvi1 ცვლადი. თუ using namespace ბლოკში გამოვაცხადებთ Sivrce_2 სივრცეს, მაშინ კომპილატორი ვერ გაარკვევს თუ სახელების რომელ სივრცეს ეკუთვნის ricxvi1 ცვლადი და გასცემს შეტყობინებას შეცდომის შესახებ. იმისათვის, რომ მივმართოთ Sivrce_2 სივრცის ricxvi1 ცვლადს, მისი სახელის წინ უნდა მივუთითოთ Sivrce_2:: პრეფიქსი.

ჩადგმული სახელების სივრცე

ჩადგმულია სახელების სივრცე, რომელიც სახელების სხვა სივრცეშია მოთავსებული. ჩადგმული სახელების სივრცის გამოყენებით შეგვიძლია შევქმნათ სახელების სივრცის იერარქია. სახელების სივრცის იერარქია შეიძლება გადანაწილებული იყოს სხვადასხვა პროგრამაში (სხვადასხვა ფაილში). ასეთი გადანაწილების უპირატესობა ისაა, რომ პროგრამისტებს შეუძლიათ ცალცალკე იმუშაონ ამ პროგრამებზე და შემდეგ ერთდროულად შეასრულონ მათი კომპილირება. მოცემული პროგრამით სრულდება ჩადგულ სახელების სივრცესთან მუშაობის დემონსტრირება.

```
//          ჩადგულ სახელების სივრცესთან მუშაობა
#pragma once

namespace Sivrce_3
{
int ricxvi7;
namespace Sivrce_31
{
int ricxvi6;
}
namespace Sivrce_32
{
int ricxvi6;
}
}
//
namespace Saxelebis_Sivrce {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
```

```
        using namespace Sivrce_3;
. . .
//
private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e) {
Sivrce_3::Sivrce_31::ricxvi6 = Convert::ToInt32(textBox1->Text);
Sivrce_3::Sivrce_32::ricxvi6 = Convert::ToInt32(textBox2->Text);
ricxvi7 = Convert::ToInt32(textBox3->Text);
label1->Text = Sivrce_3::Sivrce_31::ricxvi6.ToString() + "   " + Sivrce_3::Sivrce_32::ricxvi6.ToString() +
                "   " + ricxvi7.ToString();
}
```

Sivrce_3 სახელების სივრცეში განსაზღვრულია სახელების ორი სივრცე: Sivrce_31 და Sivrce_32. რადგან using namespace ბლოკში მითითებულია Sivrce_3 სახელების სივრცე, ამიტომ ხილული იქნება მხოლოდ ricxvi7 ცვლადი. Sivrce_31 და Sivrce_32 სივრცეში გამოცხადებული ცვლადები ხილული არ იქნება და მათთან მიმართვისათვის შესაბამისად უნდა მივუთითოთ Sivrce_3::Sivrce_31:: და Sivrce_3::Sivrce_32:: პრეფიქსები.

ახლა using namespace ბლოკში მივუთითოთ დირექტივა:

```
using namespace Sivrce_3::Sivrce_31;
```

მაშინ ჩვენი პროგრამა მიიღებს შემდეგ სახეს:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ricxvi6 = Convert::ToInt32(textBox1->Text);
Sivrce_3::Sivrce_32::ricxvi6 = Convert::ToInt32(textBox2->Text);
ricxvi7 = Convert::ToInt32(textBox3->Text);
label1->Text = ricxvi6.ToString() + "   " + Sivrce_3::Sivrce_32::ricxvi6.ToString() + "   " +
ricxvi7.ToString();
}
```

რადგან Sivrce_31 სივრცე გახდა ხილული, ამიტომ ricxvi6 ცვლადის წინ აღარ არის საჭირო Sivrce_3::Sivrce_31:: პრეფიქსის მითითება.

ახლა using namespace ბლოკში using namespace Sivrce_3; სტრიქონი წავშალოთ და დავტოვოთ using namespace Sivrce_3::Sivrce_31; სტრიქონი. ამ შემთხვევაში, Sivrce_3 სივრცე აღარ იქნება ხილული. ხილული იქნება Sivrce_31 სივრცე. ამ შემთხვევაში, ჩვენი პროგრამა შემდეგ სახეს მიიღებს:

```
#pragma once

namespace Sivrce_3
{
int ricxvi7;
namespace Sivrce_31
{
int ricxvi6;
}
namespace Sivrce_32
{
int ricxvi6;
}
}
//
namespace Saxelebis_Sivrce {
```

```
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
        using namespace Sivrce_3::Sivrce_31;
. . .
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
ricxvi6 = Convert::ToInt32(textBox1->Text);
Sivrce_3::Sivrce_32::ricxvi6 = Convert::ToInt32(textBox2->Text);
Sivrce_3::ricxvi7 = Convert::ToInt32(textBox3->Text);
label1->Text = ricxvi6.ToString() + "   " + Sivrce_3::Sivrce_32::ricxvi6.ToString() + "   "
                + Sivrce_3::ricxvi7.ToString();
}
```

სახელების სივრცეში შეგვიძლია განვსაზღვროთ კლასებიც. ქვემოთ მოცემულ პროგრამაში განსაზღვრულია Bmw და Opel სახელების სივრცე. ორივე სახელების სივრცე ცალკე ფაილში უნდა მოვათავსოთ (როგორც ამას კლასების შემთხვევაში ვაკეთებდით), რადგან ისინი კლასებს შეიცავენ.

```
//      სახელების სივრცესთან მუშაობის დემონსტრირება
//      Sivrce.h ფაილი
//      Bmw სახელების სივრცის გამოცხადება
namespace Bmw
{
public ref class Manqana
{
public :
        int ricxvi1;
        System::String^ ManqanisMarka;
};
}
//      Opel სახელების სივრცის გამოცხადება
namespace Opel
{
public ref class Manqana
{
public :
        int ricxvi1;
        System::String^ ManqanisMarka;
};
}
//      Form1.h ფაილი
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Bmw::Manqana^ ChemiManqana1 = gcnew Bmw::Manqana;
ChemiManqana1->ManqanisMarka = L"BMW";
label1->Text = ChemiManqana1->ManqanisMarka->ToString();
```

```
Opel::Manqana^ ChemiManqana2 = gcnew Opel::Manqana;
ChemiManqana2->ManqanisMarka = L"Opel";
label2->Text = ChemiManqana2->ManqanisMarka->ToString();
}
```

პროგრამაში განსაზღვრულია ორი კლასი, რომლებსაც ერთნაირი სახელები აქვთ - Manqana. იმისათვის, რომ ეს ორი კლასი ერთმანეთისაგან განვასხვავოთ, საჭიროა კლასის სახელის წინ სახელების სივრცის მითითება. მაგალითად, მოცემულ სტრიქონში ხდება იმ Manqana კლასის ობიექტის შექმნა, რომელიც BMW სახელების სივრცეშია გამოცხადებული:

```
Bmw::Manqana^ ChemiManqana1 = gcnew Bmw::Manqana;
```

ობიექტის შექმნის შემდეგ არ არის აუცილებელი სახელების სივრცის მითითება ობიექტის სახელის ან მისი წევრის წინ, მაგალითად,

```
ChemiManqana1->ManqanisMarka = L"BMW";
```

ჩადგმული შეიძლება იყოს, აგრეთვე კლასის შემცველი სახელების სივრცეც. მოცემული პროგრამით ხდება კლასის შემცველ ჩადგმულ სახელების სივრცესთან მუშაობის დემონსტრირება.

```
//      ჩადგმულ სახელების სივრცესთან მუშაობის დემონსტრირება
namespace Saxelebis_Sivrce
{
//      Chadgmuli_Sivrce1 სახელების სივრცე არის ჩადგმული
namespace Chadgmuli_Sivrce1
{
public ref class ChemiKlasi
{
public :
        System::String^ Funqcia()
{
return " Chadgmuli_Sivrce1 ";
}
};
}
//      Chadgmuli_Sivrce2 სახელების სივრცე არის ჩადგმული
namespace Chadgmuli_Sivrce2
{
public ref class ChemiKlasi
{
public :
        System::String^ Funqcia()
{
return " Chadgmuli_Sivrce2 ";
}
};
}
}
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
Saxelebis_Sivrce::Chadgmuli_Sivrce1::ChemiKlasi^ obieqti1 = gcnew
                                        Saxelebis_Sivrce::Chadgmuli_Sivrce1::ChemiKlasi;
```

```
Saxelebis_Sivrce::Chadgmuli_Sivrce2::ChemiKlasi^ obieqti2 = gcnew
                                        Saxelebis_Sivrce::Chadgmuli_Sivrce2::ChemiKlasi;
label1->Text = obieqti1->Funqcia();
label2->Text = obieqti2->Funqcia();
}
```

უსახელო სახელების სივრცე

არსებობს უსახელო სახელების სივრცე (unnamed namespace). მისი საშუალებით იქმნება ფაილის შიგნით უნიკალური იდენტიფიკატორები. უსახელო სახელების სივრცის სინტაქსია:

```
namespace {
//      კოდი
}
```

მის წევრებთან მიმართვა შესაძლებელია მათი სახელით. ფაილის გარეთ ასეთი იდენტიფიკატორები უცნობია. თუ გვინდა, რომ გლობალური ცვლადის ხილვადობის უბანი შევზღუდოთ მხოლოდ იმ ფაილით, რომელშიც ის არის გამოცხადებული, უნდა გამოვიყენოთ უსახელო სახელების სივრცე. მაგალითი.

პირველი ფაილი:

```
namespace {
int ricxvi;
}
void Funqcia() {
ricxvi = 75;
}
```

მეორე ფაილი:

```
extern int ricxvi;
void Funqcia() {
ricxvi = 20;                //      შეცდომა
}
```

თავი 10. ინფორმაციის შეტანა-გამოტანა

შეტანა-გამოტანის ნაკადების კლასები

შეტანა-გამოტანა .NET გარემოში ეფუძნება ორ ერთმანეთთან დაკავშირებულ ცნებას - ***ნაკადებსა*** და ***მონაცემების სათავსს***. ნაკადი არის მონაცემების მიმდევრობა. მონაცემების სათავსი არის ადგილი, სადაც მონაცემები ინახება. მონაცემების სათავსი შეიძლება იყოს ფაილი, ქსელური შეერთება, მისამართი ინტერნეტში და მეხსიერების უბანი. არსებობს ნაკადების შემდეგი კლასები: Stream, FileStream, NetworkStream, MemoryStream, BufferedStream და CryptoStream. ჩვენ დაწვრილებით განვიხილავთ FileStream კლასს.

C++ პროგრამები მონაცემების შეტანა-გამოტანას ნაკადების საშუალებით ახდენენ. ნაკადში იგულისხმება მონაცემების მიმდევრობა. ნაკადი ფიზიკურ მოწყობილობას შეტანა-გამოტანის სისტემის საშუალებით უკავშირდება.

C++ ენაში შეტანა-გამოტანის ოპერაციები ყველაზე დაბალ დონეზე - ბაიტებზე ოპერირებენ, რადგან შეტანა-გამოტანის მოწყობილობების უმრავლესობა არის ბაიტ-ორიენტირებული. მეორე მხრივ, კომპიუტერთან ურთიერთობისას ადამიანისათვის მოხერხებულია სიმბოლოებთან მუშაობა. ამიტომ, შეტანა-გამოტანის ოპერაციების შესრულებისას საჭირო ხდება ბაიტების გარდაქმნა სიმბოლოებად და პირიქით.

არსებობს სიმბოლოების ASCII და Unicode ნაკრები. ASCII სიმბოლოს ზომაა 1 ბაიტი, ხოლო Unicode სიმბოლოს ზომა კი - 2 ბაიტი. გავიხსენოთ, რომ C++ ენაში Char ტიპის სიმბოლო იკავებს 2 ბაიტს, Byte (unsigned char) ტიპის მნიშვნელობა კი - 1 ბაიტს. ამიტომ, ბაიტების ნაკადების გამოყენება არც ისე მოხერხებულია სიმბოლოების შეტანა-გამოტანის შესრულებისას. ამ ამოცანის გადასაწყვეტად C++ ენაში განსაზღვრულია რამდენიმე კლასი, რომლებიც ახდენენ ბაიტების ნაკადების გარდაქმნას სიმბოლოების ნაკადებად, ე.ი. ავტომატურად გარდაქმნიან Byte ტიპის მონაცემებს Char ტიპის მონაცემებად და პირიქით.

C++ ენაში არსებობს ბაიტებისა და სიმბოლოების ნაკადების კლასები, რომლებიც განსაზღვრულია System::IO კლასში. ამიტომ, ნებისმიერი პროგრამის დასაწყისში, რომელიც იყენებს ნაკადების კლასებს, უნდა ჩავრთოთ using namespace System::IO; დირექტივა.

ბაიტების ნაკადები

C++ ენაში ნაკადები იქმნება System::IO::Stream აბსტრაქტული კლასის საშუალებით. ის არის საბაზო ყველა სხვა კლასებისათვის და მასში განსაზღვრულია ნაკადებზე სტანდარტული ოპერაციები. 9.1 ცხრილში მოცემულია Stream კლასში განსაზღვრული რამდენიმე ხშირად გამოყენებადი ფუნქცია.

Stream კლასში განსაზღვრულია ფუნქციები, რომლებიც განკუთვნილია მონაცემების ჩაწერისა და წაკითხვისათვის. მაგრამ, ყველა ნაკადი არ უზრუნველყოფს მონაცემების ჩაწერისა და წაკითხვის ოპერაციებს, აგრეთვე, პოზიციონირებას Seek() ფუნქციის გამოყენებით. კონკრეტული ნაკადის შესაძლებლობების გასაგებად შეგვიძლია გამოვიყენოთ Stream კლასის თვისებები (9.2 ცხრილში).

Stream კლასს აქვს ბაიტების ნაკადების სამი მემკვიდრე კლასი, რომლებიც მოცემულია 9.3 ცხრილში.

ცხრილი 9.1. Stream კლასში განსაზღვრული ზოგიერთი ფუნქცია

ფუნქცია	აღწერა
void Close()	ნაკადს ხურავს
void Flush()	ნაკადის შემცველობას ფიზიკურ მოწყობილობაზე წერს
int ReadByte()	შესასვლელი ნაკადიდან კითხულობს 1 ბაიტს და გასცემს მის მთელრიცხვა წარმოდგენას
int Read(array<Byte>^ მასივი, int პოზიცია, int ბაიტების_რაოდენობა)	შესასვლელი ნაკადიდან კითხულობს მითითებული რაოდენობის ბაიტს და მათ ათავსებს მასივში დაწყებული მითითებული პოზიციიდან. გაიცემა წარმატებით წაკითხული ბაიტების რაოდენობა
long long Seek(long long პოზიცია, SeekOrigin გადათვლის_დასაწყისი)	გასცემს ნაკადში მიმდინარე ბაიტის პოზიციას, გადათვლილს მითითებული პოზიციიდან
void WriteByte(Byte ცვლადი)	გამოსასვლელ ნაკადში წერს ერთ ბაიტს
void Write(array<Byte>^ მასივი, int პოზიცია, int ბაიტების_რაოდენობა)	გამოსასვლელ ნაკადში მასივიდან წერს მითითებული რაოდენობის ბაიტს, დაწყებული მითითებული პოზიციიდან.

ცხრილი 9.2. Stream კლასის თვისებები

თვისება	აღწერა
bool CanRead	თვისებას აქვს true მნიშვნელობა, თუ შესაძლებელია ნაკადიდან წაკითხვა. თვისება განკუთვნილია მხოლოდ წასაკითხად.
bool CanSeek	თვისებას აქვს true მნიშვნელობა, თუ ნაკადის ელემენტის მიმდინარე პოზიციის განსაზღვრა შესაძლებელია. თვისება განკუთვნილია მხოლოდ წასაკითხად.
bool CanWrite	თვისებას აქვს true მნიშვნელობა, თუ შესაძლებელია ნაკადში ჩაწერა. თვისება განკუთვნილია მხოლოდ წასაკითხად.
long long Length	თვისება განსაზღვრავს ნაკადის სიგრძეს. თვისება განკუთვნილია მხოლოდ წასაკითხად.
long long Position	თვისება განსაზღვრავს ნაკადის მიმდინარე ელემენტის პოზიციას. თვისება განსაზღვრულია როგორც წაკითხვის, ისე ჩაწერისათვის.

ცხრილი 9.3. Stream კლასის მემკვიდრე კლასები

კლასი	აღწერა
BufferedStream	უზრუნველყოფს ბაიტების ნაკადის ბუფერიზებას
FileStream	ბაიტების ნაკადია, განკუთვნილი ფაილში შეტანა-გამოტანის ოპერაციების შესასრულებლად
MemoryStream	ბაიტების ნაკადია, რომელიც გამოიყენება მეხსიერებასთან სამუშაოდ

ცხრილი 9.4. TextReader კლასის შეტანის ფუნქციები

ფუნქცია	აღწერა
void Close()	ხურავს შესასვლელ ნაკადს
int Peek()	კითხულობს შესასვლელი ნაკადის მომდევნო სიმბოლოს, მაგრამ არ შლის მას. გასცემს -1-ს თუ არ არის მომდევნო სიმბოლო
int Read()	კითხულობს შესასვლელი ნაკადის სიმბოლოს და გასცემს მის მთელრიცხვა წარმოდგენას. გასცემს -1-ს ნაკადის დასასრულის მიღწევისას
int Read(array<Char>^ მასივი, int პოზიცია, int სიმბოლოების_რაოდენობა)	შესასვლელი ნაკადიდან კითხულობს მითითებული რაოდენობის სიმბოლოს და ათავსებს მათ მასივში დაწყებული მითითებული პოზიციიდან. გაიცემა წარმატებით წაკითხული სიმბოლოების რაოდენობა
int ReadBlock(array<Char>^ მასივი, int პოზიცია, int სიმბოლოების_რაოდენობა)	შესასვლელი ნაკადიდან კითხულობს მითითებული რაოდენობის სიმბოლოს და ათავსებს მათ მასივში დაწყებული მითითებული პოზიციიდან. ამასთან, გაიცემა წარმატებით წაკითხული სიმბოლოების რაოდენობა
String^ ReadLine()	კითხულობს ტექსტის სტრიქონს და გასცემს მას C++ ენის სტრიქონის სახით. ნულოვანი მნიშვნელობა იმ შემთხვევაში გაიცემა, თუ წაკითხული იქნა ფაილის დასასრულის სიმბოლო
String^ ReadToEnd()	კითხულობს ნაკადის დარჩენილ სიმბოლოებს და გასცემს მათ C# ენის სტრიქონის სახით

ცხრილი 9.5. TextWriter კლასის გამოტანის ფუნქციები

ფუნქცია	აღწერა
void Write(int ცვლადი)	int ტიპის მნიშვნელობის ჩაწერა
void Write(double ცვლადი)	double ტიპის მნიშვნელობის ჩაწერა
void Write(bool ცვლადი)	bool ტიპის მნიშვნელობის ჩაწერა
void WriteLine(String^ ცვლადი)	სტრიქონის ჩაწერა, რომელიც მთავრდება მომდევნო სტრიქონზე გადასვლის სიმბოლოთი
void WriteLine(uint ცვლადი)	uint ტიპის მნიშვნელობისა და მომდევნო სტრიქონზე გადასვლის სიმბოლოს ჩაწერა
void WriteLine(Char ცვლადი)	სიმბოლოს ჩაწერა, რომელსაც მოსდევს მომდევნო სტრიქონზე გადასვლის სიმბოლო
void Close()	ნაკადს ხურავს
void Flush()	ახდენს ბუფერში დარჩენილი მონაცემების ჩაწერას გამოსასვლელ ნაკადში

ცხრილი 9.6. სიმბოლოების ნაკადების კლასები

ნაკადის კლასი	აღწერა
StreamReader	ბაიტების ნაკადიდან კითხულობს სიმბოლოებს
StreamWriter	სიმბოლოებს წერს ბაიტების ნაკადში
StringReader	სტრიქონიდან კითხულობს სიმბოლოებს
StringWriter	სტრიქონში წერს სიმბოლოებს

სიმბოლოების ნაკადი

სიმბოლოების ნაკადთან სამუშაოდ გამოიყენება TextReader და TextWriter აბსტრაქტული კლასები. მათში განსაზღვრული ფუნქციები უზრუნველყოფენ შეტანა-გამოტანის ოპერაციებს სიმბოლოების ნაკადებისათვის.

9.4 ცხრილში მოცემულია TextReader კლასის შეტანის ფუნქციები. აღვნიშნოთ, რომ ReadLine() ფუნქციის გამოყენება მოსახერხებელია შესასვლელი ნაკადიდან იმ მონაცემების წასაკითხად, რომელიც ინტერვალებს შეიცავს. 9.5 ცხრილში მოცემულია TextWriter კლასში განსაზღვრული Write() და WriteLine() ფუნქციების სხვადასხვა ვერსიები.

TextReader და TextWriter კლასების რეალიზება ხდება 9.6 ცხრილში მოცემული სიმბოლოების ნაკადების კლასების საშუალებით. ეს ნაკადები უზრუნველყოფს TextReader და TextWriter კლასებში განსაზღვრულ ფუნქციებსა და თვისებებს.

ორობითი ნაკადი

C++ ენაში განსაზღვრულია აგრეთვე, ორობითი ნაკადის ორი კლასი BinaryReader და BinaryWriter. ისინი შეგვიძლია გამოვიყენოთ ორობითი მონაცემების ჩაწერისა და წაკითხვისათვის.

FileStream კლასი

ეს კლასი გამოიყენება ფაილში ბაიტ-ორიენტირებული შეტანა-გამოტანის შესასრულებლად ანუ ფაილში ბაიტების ჩაწერა-წაკითხვის ოპერაციების შესასრულებლად. რადგან ფაილის ყველაზე გავრცელებული ტიპია დისკური ფაილი, ამიტომ შემდგომში ამ ტიპის ფაილებს განვიხილავთ.

ფაილის გახსნა და დახურვა

ფაილთან დაკავშირებული ბაიტების ნაკადის შესაქმნელად საჭიროა FileStream ობიექტის ფორმირება. მის ფორმირებას ახდენს კონსტრუქტორი, რომლის სინტაქსია:

FileStream(String^ *ფაილის_სახელი,* FileMode::*რეჟიმი*);

სადაც, *ფაილის_სახელი* არის გასახსნელი ფაილის სახელი, რომელიც შეიძლება შეიცავდეს, აგრეთვე, ამ ფაილისაკენ გზას, მაგალითად "C:\\My Documents\\A1\\File1.txt". *რეჟიმი* პარამეტრი განსაზღვრავს თუ როგორ უნდა გაიხსნას ფაილი. მისი მნიშვნელობები ანუ ფაილის გახსნის რეჟიმები მოცემულია 9.7 ცხრილში.

ქვემოთ მოცემული პროგრამით სრულდება testfile.dat ფაილის გახსნა და დახურვა. არ უნდა დაგვავიწყდეს პროგრამის დასაწყისში using namespace ბლოკში

using namespace System::IO;

დირექტივის ჩართვა. ეს უნდა გავაკეთოთ ამ თავში მოცემული ყველა პროგრამისათვის.

```
{
//      ფაილის გახსნა და დახურვა
FileStream^ file_in;

file_in = gcnew FileStream("filetest.txt", FileMode::Open);          //      ფაილის გახსნა
file_in->Close();                                                    //      ფაილის დახურვა
}
```

ცხრილი 9.7. FileMode პარამეტრის მნიშვნელობები

მნიშვნელობა	აღწერა
FileMode::Append	მონაცემები დაემატება ფაილის ბოლოში
FileMode::Create	იქმნება ახალი გამოსასვლელი ფაილი. ამავე სახელის მქონე ადრე შექმნილი ფაილი იშლება
FileMode::CreateNew	იქმნება ახალი გამოსასვლელი ფაილი
FileMode::Open	იხსნება არსებული ფაილი
FileMode::OpenOrCreate	თუ ფაილი არსებობს, მაშინ ის გაიხსნება, თუ არ არსებობს, მაშინ ის შეიქმნება
FileMode::Truncate	იხსნება არსებული ფაილი, მაგრამ მისი სიგრძე ჩამოიჭრება ნულამდე

პროგრამის პირველი ორი სტრიქონი შეგვიძლია შევცვალოთ ერთით:

```
FileStream^ file_in = gcnew FileStream("filetest.dat", FileMode::Open);
```

პროგრამაში მითითებულია ფაილის მხოლოდ სახელი, ამიტომ მისი ძებნა შესრულდება მიმდინარე კატალოგში. თუ ფაილი სხვა კატალოგშია მოთავსებული, მაშინ უნდა მივუთითოთ გზა ამ ფაილისაკენ:

```
file_in =
gcnew FileStream("C:\\My Documents\\Visual Studio Projects\\ A1\\filetest.txt", FileMode::Open);
```

თუ ფაილთან მიმართვა უნდა შეიზღუდოს მხოლოდ წაკითხვით ან ჩაწერით, მაშინ უნდა მივუთითოთ ფაილთან მიმართვის სახე:

FileStream(String^ *ფაილის_სახელი,* FileMode::*რეჟიმი,* FileAccess::*მიმართვის_სახე)

აქ *მიმართვის_სახე* იღებს მნიშვნელობებს: FileAccess::Read, FileAccess::Write და FileAccess::ReadWrite. მაგალითად,

```
FileStream^ file_in = gcnew FileStream("file1.txt",FileMode::Open,FileAccess::Read);
```

აქ file1.txt ფაილი იხსნება წაკითხვის რეჟიმში.

ფაილთან მუშაობის დასამთავრებლად აუცილებელია მისი დახურვა Close() ფუნქციის გამოყენებით. ფაილის დახურვისას თავისუფლდება მისთვის გამოყოფილი ოპერაციული სისტემის რესურსები.

ფაილში ბაიტების ჩაწერა

FileStream კლასში განსაზღვრულია ორი ფუნქცია, რომლებიც ფაილში წერენ ბაიტებს: WriteByte() და Write(). WriteByte() ფუნქცია გამოიყენება ფაილში ბაიტის ჩასაწერად. მისი სინტაქსია:

void WriteByte(Byte *ცვლადის_სახელი*)

Write() ფუნქცია გამოიყენება ფაილში ბაიტების მასივის ჩასაწერად. მისი სინტაქსია:

void Write(array<Byte>^ *ბუფერი,* int *პოზიცია,* int *ბაიტების_რაოდენობა*)

Write() ფუნქცია ბუფერიდან (მასივიდან) ფაილში ჩაწერს მითითებული რაოდენობის ბაიტს, დაწყებული მითითებული პოზიციიდან.

ფაილში ჩაწერისას, ხშირად არ ხდება მონაცემების მაშინვე ჩაწერა ფიზიკურ მოწყობილობაზე. ამის ნაცვლად, ოპერაციული სისტემა ახდენს გამოტანის ბუფერიზებას ანუ ჩასაწერ ბაიტებს ბუფერში აგროვებს. ბუფერი არის მეხსიერების გარკვეული უბანი. ბუფერიზება სრულდება მანამ, სანამ არ დაგროვდება მონაცემების საკმარისი რაოდენობა. ბუფერის შევსებისთანავე მონაცემები ჩაიწერება დისკზე. შედეგად, იზრდება ოპერაციული სისტემის მწარმოებლურობა. საქმე ის არის, რომ დისკზე მონაცემების ჩაწერა ხდება სექტორებად, რომელთა ზომები იცვლება 128 ბაიტიდან დაწყებული ზევით. გამოტანის

შედეგების ბუფერიზება ხდება მანამ, სანამ დისკზე შესაძლებელი არ გახდება მთელი სექტორის ჩაწერა.

ზოგჯერ, საჭიროა მონაცემების დისკზე ჩაწერა მიუხედავად იმისა, ბუფერი შევსებულია თუ არა. ასეთ შემთხვევაში, უნდა გამოვიყენოთ Flush() ფუნქცია. მისი სინტაქსია:

void Flush()

ფაილთან მუშაობის დამთავრებისას ის უნდა დაიხუროს Close() ფუნქციის გამოყენებით. ამ დროს ბუფერში მოთავსებული მონაცემები, დისკზე აუცილებლად ჩაიწერება. ამიტომ, Close() ფუნქციის გამოძახების წინ არაა საჭირო Flush() ფუნქციის გამოძახება.

მოცემულ პროგრამაში Write() ფუნქციის გამოიყენებით სრულდება filetext.txt ფაილში ბაიტების ჩაწერა და მათი ეკრანზე გამოტანა.

```
{
//      მასივიდან ფაილში ბაიტების ჩაწერა Write() ფუნქციის გამოიყენებით
array<Byte>^ masivi = gcnew array<Byte>(10) { 10, 20, 30, 40, 50 };
FileStream^ file_out;
file_out = gcnew FileStream("d_file.txt", FileMode::Create);
//      ბაიტების ჩაწერა d_file.txt ფაილში
file_out->Write(masivi, 2, 4);

file_out->Close();
}
```

პროგრამის შესრულების შედეგად filetext.txt ფაილში ჩაიწერება masivi მასივის 4 ელემენტი დაწყებული მესამე ელემენტიდან (masivi[2]). თუ გვინდა მთელი მასივის ჩაწერა ფაილში, მაშინ უნდა მივუთითოთ file_out->Write(masivi,0,masivi->Length);

მოცემული პროგრამით ხდება ბაიტების ჩაწერა ფაილში WriteByte() ფუნქციის გამოყენებით:

```
{
//      მასივიდან ფაილში ბაიტების ჩაწერა WriteByte() ფუნქციის გამოიყენებით
array<Byte>^ masivi = gcnew array<Byte>(10) { 10, 20, 30, 40, 50 };
FileStream^ file_out;
file_out = gcnew FileStream("d_file.txt", FileMode::Create);

//      ბაიტების ჩაწერა d_file.txt ფაილში
for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        file_out->WriteByte(masivi[indexi]);

file_out->Close();
}
```

ფაილიდან ბაიტების წაკითხვა

FileStream კლასში განსაზღვრულია ორი ფუნქცია, რომლებიც ფაილიდან ბაიტებს კითხულობენ: ReadByte() და Read(). ReadByte() ფუნქციის სინტაქსია:

int ReadByte()

ამ ფუნქციის გამოძახებისას ხდება ერთი ბაიტის წაკითხვა ფაილიდან და გაიცემა ბაიტის კოდი (მთელრიცხვა მნიშვნელობა). თუ წაკითხულ იქნა ფაილის დასასრულის სიმბოლო (EOF, End Of File), მაშინ გაიცემა -1.

ბაიტების ბლოკის წასაკითხად უნდა გამოვიყენოთ Read() ფუნქცია. მისი სინტაქსია:

int Read(array<Byte>^ *ბუფერი*, int *პოზიცია*, int *ბაიტების_რაოდენობა*)

Read() ფუნქცია ცდილობს ფაილიდან წაიკითხოს მითითებული რაოდენობის ბაიტი და მოათავსოს ისინი ბუფერში დაწყებული მითითებული პოზიციიდან. გაიცემა რეალურად წაკითხული ბაიტების რაოდენობა.

მოცემული პროგრამით სრულდება ბაიტების წაკითხვა ReadByte() ფუნქციის გამოყენებით და მათი ეკრანზე გამოტანა. დავუშვათ, filetext.txt ფაილში ჩაწერილია ციფრები: 1234567890 .

```
{
//          ბაიტების წაკითხვის დემონსტრირება ReadByte() ფუნქციის გამოყენებით
int ricxvi;
FileStream^ file1 = gcnew FileStream("d_file.txt", FileMode::Open);

// მონაცემების წაკითხვა ფაილიდან მანამ, სანამ არ შეგვხვდება ფაილის დასასრული
for ( ; ricxvi != -1; )                 //          თუ ricxvi = -1, მაშინ მიღწეულია ფაილის დასასრული
{
ricxvi = file1->ReadByte();
if ( ricxvi != -1 ) label1->Text += (Char) ricxvi;
}
file1->Close();
}
```

მოცემული პროგრამით სრულდება ბაიტების წაკითხვა Read() ფუნქციის გამოყენებით და მათი ეკრანზე გამოტანა. დავუშვათ, filetext.txt ფაილში ჩაწერილია ციფრები: 1234567890 .

```
{
//          ბაიტების წაკითხვის დემონსტრირება Read() ფუნქციის გამოყენებით
int wakitxuli_baitebis_raodenoba;
int pozicia   = Convert::ToInt32(textBox1->Text),
raodenoba = Convert::ToInt32(textBox2->Text);
array<Byte>^ masivi = gcnew array<Byte>(10);
FileStream^ file1 = gcnew FileStream("d_file.txt", FileMode::Open);
label1->Text = "";
//          masivi მასივში ჩაიწერება 4 ბაიტი დაწყებული მე-3 ელემენტიდან
wakitxuli_baitebis_raodenoba = file1->Read(masivi, pozicia, raodenoba);
file1->Close();
for ( int indexi = 0; indexi < masivi->Length; indexi++ )
          label1->Text += masivi[indexi] + "   ";
label2->Text = wakitxuli_baitebis_raodenoba.ToString();
}
```

როგორც ვიცით, Read() ფუნქციას სამი არგუმენტი აქვს: ბუფერი, პოზიცია და ბაიტების რაოდენობა. შესაბამისად, პროგრამაში გამოცხადებულია masivi მასივი, pozicia და raodenoba ცვლადები. დავუშვათ, pozicia ცვლადს მივანიჭეთ 2, raodenoba ცვლადს კი - 4. შედეგად, filetext.txt ფაილიდან შესრულდება 4 ბაიტის წაკითხვა და masivi მასივში მესამე პოზიციიდან (masivi[2]) ჩაწერა. masivi მასივი მიიღებს სახეს:

0 0 49 50 51 52 0 0 0 0

თუ გვინდა, რომ ბაიტების ნაცვლად სიმბოლოები გამოჩნდეს უნდა შევასრულოთ ტიპის გარდაქმნა:

```
label1->Text += (Char)masivi[indexi] + "   ";
```

Read() ფუნქცია, აგრეთვე, გასცემს რეალურად წაკითხული ბაიტების რაოდენობას,

რომელიც wakitxuli_baitebis_raodenoba ცვლადში მოთავსთდება.

მოცემული პროგრამით სრულდება ერთი ფაილის მეორეში გადაწერა.

```
{
//      ერთი ფაილის მეორეში გადაწერა
label1->Text = "";
int ricxvi;
FileStream^ file_in;
FileStream^ file_out;
//      საწყისი ფაილის გახსნა
file_in = gcnew FileStream("d_file.txt", FileMode::Open);
//      მიმღები ფაილის გახსნა
file_out = gcnew FileStream("s_file.txt", FileMode::Create);
//      ბაიტების გადაწერა file_in ფაილიდან file_out ფაილში
for ( ; ricxvi != -1; )
{
//      ბაიტების წაკითხვა file_in ფაილიდან
ricxvi = file_in->ReadByte();
//      ბაიტების ჩაწერა file_out ფაილში
if ( ricxvi != -1 ) file_out->WriteByte( ( Byte ) ricxvi );
label1->Text += (Char) ricxvi + "   ";
}
//      ფაილების დახურვა
file_in->Close();
file_out->Close();
}
```

ფაილში სიმბოლოების შეტანა-გამოტანა

ფაილში სიმბოლოების ჩასაწერად და წასაკითხად სიმბოლოების ნაკადები გამოიყენება. ისინი Unicode სიმბოლოებზე ოპერირებენ. სიმბოლურ ფაილებთან მუშაობისას FileStream ობიექტი უნდა ჩავრთოთ StreamReader ან StreamWrtiter კლასის შემადგენლობაში. ეს კლასები უზრუნველყოფს ბაიტების ნაკადების სიმბოლოების ნაკადებად ავტომატურ გარდაქმნას და პირიქით.

StreamWriter კლასი არის TextWriter კლასის მემკვიდრე. StreamReader კლასიც TextReader კლასის მემკვიდრეა. შედეგად, StreamWriter და StreamReader კლასების წევრებს შეუძლიათ მიმართონ TextWriter და TextReader კლასების ფუნქციებსა და თვისებებს.

StreamWriter კლასი

სიმბოლოების გამოსასვლელი ნაკადი გამოიყენება სიმბოლოების ფაილში მონაცემების ჩასაწერად. ამისათვის FileStream ობიექტი უნდა ჩავრთოთ StreamWriter კლასის შემადგენლობაში. ამ კლასში განსაზღვრულია რამდენიმე კონსტრუქტორი. ხშირად გამოყენებადი კონსტრუქტორის სინტაქსია:

StreamWriter(Stream^ *ნაკადის_სახელი*)

ობიექტის შექმნისთანავე StreamWriter კლასი ავტომატურად გარდაქმნის სიმბოლოებს ბაიტებად. მოცემული პროგრამა ახდენს textBox კომპონენტში შეტანილი ტექსტის ჩაწერას

ფაილში.

```
{
//      ფაილში ჩაწერის დემონსტრირება StreamWriter კლასის გამოყენებით
String^ striqoni;
FileStream^ file_out = gcnew FileStream("text.txt", FileMode::Create);
//
StreamWriter^ file_stream_out = gcnew StreamWriter(file_out);

//      textBox კომპონენტში შეტანილი ტექსტის მინიჭება striqoni ცვლადისათვის
striqoni = textBox1->Text;              //      შევიტანოთ - რომან სამხარაძე
file_stream_out->Write(striqoni);       //      text.txt ფაილში striqoni სტრიქონის ჩაწერა
file_stream_out->Close();
}
```

StreamReader კლასი

სიმბოლოების შესასვლელი ნაკადი გამოიყენება სიმბოლოების ნაკადიდან მონაცემების წასაკითხად. ამისათვის FileStream ობიექტი უნდა ჩავრთოთ StreamReader კლასის შემადგენლობაში. ამ კლასში განსაზღვრულია რამდენიმე კონსტრუქტორი. ხშირად გამოყენებადი კონსტრუქტორის სინტაქსია:

StreamReader(Stream^ *ნაკადის_სახელი*)

StreamReader ობიექტის შექმნისთანავე ავტომატურად სრულდება ბაიტების გარდაქმნა სიმბოლოებად. მოცემული პროგრამა კითხულობს ტექსტურ ფაილს და მისი შემცველობა ეკრანზე გამოაქვს.

```
{
//      ტექსტური ფაილიდან წაკითხვის დემონსტრირება
//      StreamReader კლასის გამოყენებით
String^ striqoni;
FileStream^ file_in = gcnew FileStream("text.txt", FileMode::Open );
StreamReader^ file_stream_in = gcnew StreamReader(file_in);

for (;  ( striqoni = file_stream_in->ReadLine() )  != nullptr; )
        label1->Text += striqoni + "\n";
file_stream_in->Close();
}
```

პროგრამაში ფაილიდან სტრიქონების წაკითხვა მთავრდება მაშინ, როდესაც ReadLine() ფუნქციის მიერ გაცემული სტრიქონი მიიღებს nullptr მნიშვნელობას. სწორედ, ეს მიუთითებს ფაილის დასასრულზე.

ორობითი მონაცემების წაკითხვა და ჩაწერა

ფაილში ბაიტებისა და სიმბოლოების წაკითხვისა და ჩაწერის გარდა შესაძლებელია სხვა ტიპის (ორობითი) მონაცემების, int, double და ა.შ., წაკითხვა და ჩაწერა. მათი წაკითხვისა და ჩაწერისათვის გამოიყენება BinaryReader და BinaryWriter ნაკადები. ასეთი მონაცემების წაკითხვა და ჩაწერა ხდება შიგა ორობითი ფორმატით და არა ტექსტური ფორმით.

BinaryWriter კლასი

BinaryWriter კლასი გამოიყენება ფაილში ორობითი მონაცემების ჩასაწერად. ამ კლასის ხშირად გამოყენებადი კონსტრუქტორის სინტაქსია:

BinaryWriter(Stream^ *გამოსასვლელი_ნაკადი*)

აქ *გამოსასვლელი_ნაკადი* პარამეტრი განსაზღვრავს იმ ნაკადს, რომელშიც სრულდება მონაცემების ჩაწერა. ის არის ობიექტი, რომელსაც ქმნის FileStream კონსტრუქტორი.

BynaryWriter კლასში განსაზღვრულია ფუნქციები, რომლებიც უზრუნველყოფენ C++ ენის ყველა ჩადგმული ტიპის მონაცემის ჩაწერას. ზოგიერთი ფუნქცია მოცემულია 9.8 ცხრილში. ამავე კლასში განსაზღვრულია, აგრეთვე, Close() და Flush() ფუნქციები.

ცხრილი 9.8. BinaryWriter ნაკადის ზოგიერთი ფუნქცია

ფუნქცია	აღწერა
void Write(SByte ცვლადი)	ნიშნიანი ბაიტის ჩაწერა
void Write(Byte ცვლადი)	უნიშნო ბაიტის ჩაწერა
void Write(array<byte>^ მასივი)	ბაიტების მასივის ჩაწერა
void Write(short ცვლადი)	მოკლე მთელი რიცხვის ჩაწერა
void Write(UInt16 ცვლადი)	უნიშნო მოკლე მთელი რიცხვის ჩაწერა
void Write(int ცვლადი)	მთელი რიცხვის ჩაწერა
void Write(UInt32 ცვლადი)	უნიშნო მთელი რიცხვის ჩაწერა
void Write(long long ცვლადი)	გრძელი მთელი რიცხვის ჩაწერა
void Write(UInt64 ცვლადი)	უნიშნო გრძელი მთელი რიცხვის ჩაწერა
void Write(double ცვლადი)	double ტიპის მნიშვნელობის ჩაწერა
void Write(Char ცვლადი)	სიმბოლოს ჩაწერა
void Write(array<Char>^ მასივი)	სიმბოლოების მასივის ჩაწერა
void Write(String^ ცვლადი)	სტრიქონის ჩაწერა

BinaryReader ნაკადი

BinaryReader ნაკადი გამოიყენება ფაილიდან ორობითი მონაცემის წასაკითხად. ამ კლასის ხშირად გამოყენებადი კონსტრუქტორის სინტაქსია:

BinaryReader(Stream^ *შესასვლელი_ნაკადი*)

აქ *შესასვლელი_ნაკადი* პარამეტრი განსაზღვრავს იმ ნაკადს, საიდანაც სრულდება მონაცემების წაკითხვა. ის არის ობიექტი, რომელსაც ქმნის FileStream კონსტრუქტორი.

BinaryReader კლასში განსაზღვრულია ფუნქციები, რომლებიც ახდენენ C++ ენის ყველა მარტივი ტიპის მონაცემის წაკითხვას. ხშირად გამოყენებადი ფუნქციები მოცემულია 9.9 ცხრილში. ამავე კლასში განსაზღვრულია, აგრეთვე, Read() ფუნქციის სამი ვერსია:

int Read() - კითხულობს შესასვლელი ნაკადის მომდევნო სიმბოლოს და გასცემს მის მთელრიცხვა წარმოდგენას. ფაილის დასასრულის მიღწევისას გაიცემა -1.

int Read(byte[ ] *მასივი*, int *პოზიცია*, int *ბაიტების_რაოდენობა*) - ბუფერიდან კითხულობს მითითებული რაოდენობის ბაიტს დაწყებული მითითებული პოზიციიდან. გაიცემა წარმატებით წაკითხული ბაიტების რაოდენობა.

int Read(char[ ] *მასივი*, int *პოზიცია*, int *სიმბოლოების_რაოდენობა*) - ბუფერიდან კითხულობს მითითებული რაოდენობის სიმბოლოს დაწყებული მითითებული პოზიციიდან. გაიცემა წარმატებით წაკითხული სიმბოლოების რაოდენობა.

ამავე ნაკადში განსაზღვრულია, აგრეთვე, Close() ფუნქცია.

ცხრილი 9.9. BinaryReader ნაკადის ხშირად გამოყენებადი ფუნქციები

ფუნქცია	აღწერა
bool ReadBoolean()	bool ტიპის მნიშვნელობის წაკითხვა
Byte ReadByte()	byte ტიპის მნიშვნელობის წაკითხვა
SByte ReadSByte()	sbyte ტიპის მნიშვნელობის წაკითხვა
array<Byte>^ ReadBytes(int ბაიტების_რაოდენობა)	მითითებული რაოდენობის ბაიტის წაკითხვა და მათი გაცემა მასივის სახით
Char ReadChar()	char ტიპის მნიშვნელობის წაკითხვა
array<Char>^ ReadChars(int სიმბოლოების_რაოდენობა)	მითითებული რაოდენობის სიმბოლოს წაკითხვა და მათი გაცემა მასივის სახით
double ReadDouble()	double ტიპის მნიშვნელობის წაკითხვა
float ReadSingle()	float ტიპის მნიშვნელობის წაკითხვა
short ReadInt16()	short ტიპის მნიშვნელობის წაკითხვა
int ReadInt32()	int ტიპის მნიშვნელობის წაკითხვა
long long ReadInt64()	long ტიპის მნიშვნელობის წაკითხვა
Uint16 ReadUInt16()	ushort ტიპის მნიშვნელობის წაკითხვა
UInt32 ReadUInt32()	uint ტიპის მნიშვნელობის წაკითხვა
UInt64 ReadUint64()	ulong ტიპის მნიშვნელობის წაკითხვა
String^ ReadString()	სტრიქონის წაკითხვა

ქვემოთ მოცემულ პროგრამაში გამოყენებულია BinaryReader და BinaryWriter ნაკადები. პროგრამით ჯერ სრულდება ფაილში სხვადასხვა ტიპის მონაცემების ჩაწერა, შემდეგ კი წაკითხვა.

```
{
//      ფაილში ორობითი მონაცემების ჩაწერა-წაკითხვის დემონსტრირება
BinaryWriter^ file_out;
BinaryReader^ file_in;

int mteli1 = 10, mteli2;
double wiladi1 = 1001.47, wiladi2, wiladi3;
bool b1 = true, b2;
file_out = gcnew BinaryWriter( File::Open( "file1.dat", FileMode::Create ) );

//      ფაილში მთელი რიცხვის ჩაწერა
file_out->Write(mteli1);
//      ფაილში წილადის ჩაწერა
file_out->Write(wiladi1);
//      ფაილში ლოგიკური მონაცემის ჩაწერა
file_out->Write(b1);
//      ფაილში იწერება 10.2 * 2.3 გამოსახულების გამოთვლის შედეგი
file_out->Write(10.2 * 2.3);
file_out->Close();
//      ფაილიდან წაკითხვა
file_in = gcnew BinaryReader( File::Open( "file1.dat", FileMode::Open ) );
//      მთელი რიცხვის წაკითხვა ფაილიდან
```

```
mteli2 = file_in->ReadInt32();
label1->Text = mteli2.ToString();
//        წილადის წაკითხვა ფაილიდან
wiladi2 = file_in->ReadDouble();
label2->Text = wiladi2.ToString();
//        ლოგიკური მონაცემის წაკითხვა ფაილიდან
b2 = file_in->ReadBoolean();
label3->Text = b2.ToString();
//        წილადის წაკითხვა ფაილიდან
wiladi3 = file_in->ReadDouble();
label4->Text = wiladi3.ToString();
file_in->Close();
}
```

ფაილთან პირდაპირი მიმართვა

აქამდე ვიყენებდით ფაილთან მიმდევრობით მიმართვას. ამ დროს, მიმდევრობით სრულდება ფაილში მონაცემების ჩაწერა და ფაილიდან მონაცემების წაკითხვა. ფაილთან შესაძლებელია, აგრეთვე, პირდაპირი (ნებისმიერი) მიმართვაც. ამისათვის, გამოიყენება საფაილო მიმთითებელი, რომელიც არის ფაილის მიმდინარე ელემენტის პოზიციის ნომერი. საფაილო მიმთითებლის მნიშვნელობის შესაცვლელად გამოიყენება Seek()ფუნქცია. მისი სინტაქსია:

long Seek(long *პოზიციის_ნომერი,* SeekOrigin *გადათვლის_დასაწყისი*)

პოზიციის_ნომერი პარამეტრი მიუთითებს საფაილო მიმთითებლის ახალ პოზიციას (ბაიტებით) გადათვლილს *გადათვლის_დასაწყისი* პარამეტრით განსაზღვრული ადგილიდან. *გადათვლის_დასაწყისი* პარამეტრი იღებს SeekOrigin ჩამონათვალში განსაზღვრული მნიშვნელობებიდან ერთ-ერთს (9.10 ცხრილი).

ცხრილი 9.10. გადათვლის_დასაწყისი პარამეტრის მნიშვნელობები

მნიშვნელობა	აღწერა
Begin	გადათვლა ფაილის დასაწყისიდან
Current	გადათვლა მიმდინარე პოზიციიდან
End	გადათვლა ფაილის დასასრულიდან

Seek() ფუნქციის გამოძახების შემდეგ წაკითხვის ან ჩაწერის ოპერაციები სრულდება საფაილო მიმთითებლის მიერ განსაზღვრული ახალი პოზიციიდან.

ქვემოთ მოცემული პროგრამით ხდება შეტანა-გამოტანის ოპერაციის დემონსტრირება ფაილთან ნებისმიერი მიმართვით. ფაილში ჯერ ჩაიწერება ანბანის ასოები, შემდეგ კი ხდება მათი სხვადასხვა მიმდევრობით წაკითხვა.

```
{
//        ფაილში ჩაწერა-წაკითხვის ოპერაციების შესრულება პირდაპირი მიმართვით
FileStream^ file = gcnew FileStream("file.dat", FileMode::Create);
Char simbolo;
label1->Text = "";
//        ასოების ჩაწერა ანბანის მიხედვით
for ( int i = 0; i < 26; i++ )
```

```
{
        file->WriteByte((L'a' + i));
}
//      ასოების წაკითხვა ფაილიდან
file->Seek(0, SeekOrigin::Begin);                       //      პირველი ბაიტის არჩევა
simbolo = (char) file->ReadByte();                      //      პირველი ბაიტის წაკითხვა
label1->Text += L"პირველი მნიშვნელობა    " + simbolo + "\n";
file->Seek(4, SeekOrigin::Begin);                       //      მეხუთე ბაიტის არჩევა
simbolo = (char) file->ReadByte();                      //      მეხუთე ბაიტის წაკითხვა
label1->Text += L"მეხუთე მნიშვნელობა    " + simbolo + "\n";
file->Close();
}
```

ფაილის შესახებ ინფორმაციის მიღება

System.IO სახელების სივრცეში განსაზღვრულია, აგრეთვე, File და FileInfo კლასები. File კლასში განსაზღვრულია სტატიკური ფუნქციები (9.11 ცხრილი), FileInfo კლასში კი - ჩვეულებრივი ფუნქციები (9.12 ცხრილი). რადგან File კლასის ფუნქციები სტატიკურია, ამიტომ მათთან სამუშაოდ არ არის საჭირო ობიექტის შექმნა, FileInfo კლასის ფუნქციებთან სამუშაოდ კი საჭიროა ობიექტის შექმნა. 9.13 ცხრილში მოცემულია FileInfo კლასის თვისებები.

როგორც ცხრილებიდან ჩანს, თითოეული კლასი იძლევა ფაილის შექმნის, წაშლის, გადატანის, გახსნის და ა.შ. საშუალებას. ბუნებრივია, იბადება კითხვა - თუ რომელი კლასი უნდა გამოვიყენოთ ფაილებთან სამუშაოდ. ასეთ დროს უნდა გავითვალისწინოთ რამდენიმე მოსაზრება. ჯერ ერთი, FileInfo კლასში განსაზღვრულია ისეთი ოპერაციები, რომლებიც არ არის განსაზღვრული File კლასში. მეორე, File კლასი უფრო სწრაფად მუშაობს ფაილებზე ერთი ოპერაციის შესრულებისას, რადგან დრო არ იხარჯება ობიექტის შექმნაზე. FileInfo კლასი უფრო სწრაფად მუშაობს ფაილზე რამდენიმე ოპერაციის შესრულებისას, რადგან ფაილთან მიმართვის უფლებები მოწმდება ერთხელ მისი შექმნისას, და არა მასთან ყოველი მიმართვისას.

ცხრილი 9.11. File კლასში განსაზღვრული სტატიკური ფუნქციები

ფუნქცია	აღწერა
StreamWriter^ AppendText(String^ *ფაილის_სახელი*)	არსებულ ფაილს ტექსტს უმატებს
void Copy(String^ *საწყისი_ფაილი*, String^ *მიმღები_ფაილი*)	ახდენს ფაილის გადაწერას
Filestream^ Create(String^ *ფაილის_სახელი*)	ქმნის ფაილს
StreamWriter^ CreateText(String^ *ფაილის_სახელი*)	ქმნის ფაილს და ხსნის მას ტექსტის ჩასაწერად
void Delete(String^ *ფაილის_სახელი*)	შლის ფაილს
bool Exists(String^ *ფაილის_სახელი*)	ამოწმებს ფაილის არსებობას
FileAttributes GetAttributes(String^ *ფაილის_სახელი*)	გასცემს ფაილის ატრიბუტებს
DateTime GetCreationTime(String^ *ფაილის_სახელი*)	გასცემს ფაილის შექმნის თარიღს
DateTime GetLastAccessTime(String^ *ფაილის_სახელი*)	გასცემს ფაილთან უკანასკნელი მიმართვის თარიღს
DateTime GetLastWriteTime(String^ *ფაილის_სახელი*)	გასცემს ფაილში უკანასკნელი ჩაწერის თარიღს
void Move(String^ *საწყისი_ფაილი*, String^ *მიმღები_ფაილი*)	ახდენს ფაილის გადაადგილებას
FileStream^ Open(String^ *ფაილის_სახელი*, FileMode::*რეჟიმი*)	ხსნის ფაილს
FileStream^ OpenRead(String^ *ფაილის_სახელი*)	ხსნის ფაილს წაკითხვისათვის
StreamReader^ OpenText(String^ *ფაილის_სახელი*)	ხსნის ფაილს ტექსტის წაკითხვისათვის
FileStream^ OpenWrite(String^ *ფაილის_სახელი*)	ხსნის ფაილს ჩაწერისათვის
void SetAttributes(String^ *ფაილის_სახელი*, FileAttributes:: *ფაილის_ატრიბუტები*)	აყენებს ფაილის ატრიბუტებს
void SetCreationTime(String^ *ფაილის_სახელი*, DateTime::*შექმნის_დრო*)	აყენებს ფაილის შექმნის თარიღს
void SetLastAccessTime(String^ *ფაილის_სახელი*, DateTime::*უკანასკნელი_მიმართვის_დრო*)	აყენებს ფაილთან უკანასკნელი მიმართვის თარიღს
void SetLastWriteTime(String^ *ფაილის_სახელი*, DateTime::*უკანასკნელი_ჩაწერის_დრო*)	აყენებს ფაილში უკანასკნელი ჩაწერის თარიღს

ცხრილი 9.12. FileInfo კლასში განსაზღვრული ფუნქციები

ფუნქცია	აღწერა
StreamWriter^ AppendText()	არსებულ ფაილს ტექსტს უმატებს
FileInfo^ CopyTo(String^ *მიმღები_ფაილი*)	ახდენს ფაილის გადაწერას
FileStream^ Create()	ქმნის ფაილს
StreamWriter^ CreateText()	ქმნის ტექსტურ ფაილს
void Delete()	შლის ფაილს
void MoveTo(String^ *მიმღები_ფაილი*)	ახდენს ფაილის გადატანას
FileStream^ Open(FileMode::*რეჟიმი*)	ხსნის ფაილს
StreamReader^ OpenText()	ხსნის ფაილს ტექსტის წასაშლელად
FileStream^ OpenWrite()	ხსნის ფაილს ჩაწერისათვის

ცხრილი 9.13. FileInfo კლასში განსაზღვრული თვისებები

თვისება	ტიპი	აღწერა
Attributes	FileAttributes	გასცემს ან აყენებს ფაილის ატრიბუტებს
CreationTime	DateTime	გასცემს ან აყენებს ფაილის შექმნის თარიღს
Directory	DirectoryInfo^	გასცემს კატალოგს, რომელშიც ფაილია მოთავსებული
DirectoryName	String^	გასცემს კატალოგის სახელს, რომელშიც ფაილია მოთავსებული
Exists	bool	თუ ფაილი არსებობს, მაშინ გასცემს true მნიშვნელობას, წინააღმდეგ შემთხვევაში კი false მნიშვნელობას
Extension	String^	გასცემს ფაილის გაფართოებას
FullName	String^	გასცემს ფაილისკენ გზას და ფაილის სახელს
LastAccessTime	DateTime	გასცემს ან აყენებს ფაილთან უკანასკნელი მიმართვის თარიღს
LastWriteTime	DateTime	გასცემს ან აყენებს ფაილში უკანასკნელი ჩაწერის თარიღს
Length	long long	გასცემს ფაილის ზომას
Name	String^	გასცემს ფაილის სახელს

განვიხილოთ მაგალითები. მოცემული პროგრამით ხდება File კლასის AppendText ფუნქციასთან მუშაობის დემონსტრირება.

```
{
//      File კლასის AppendText() ფუნქციასთან მუშაობის დემონსტრირება
//      ფაილისათვის ხდება სტრიქონების დამატება
String^ path = "d_file.txt";
StreamWriter^ faili = File::AppendText(path);
faili->WriteLine(L"რომან");
faili->WriteLine(L"სამხარაძე");
faili->WriteLine(L"C++ დაპროგრამების ენაა");
faili->Close();
}
```

მოცემული პროგრამით ხდება File კლასის OpenText ფუნქციასთან მუშაობის დემონსტრირება.

```
{
```

```
//      File კლასის OpenText() ფუნქციასთან მუშაობის დემონსტრირება
//      ფაილის გახსნა წაკითხვის მიზნით
StreamReader^ faili = File::OpenText("d_file.txt");
String^ striqoni = "";
for ( ; ( striqoni = faili->ReadLine() ) != nullptr; )
        label1->Text += striqoni + "   ";
}
```

მოცემული პროგრამით ხდება File კლასის Delete და Copy ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      File კლასის Delete() და Copy() ფუნქციასთან
//      მუშაობის დემონსტრირება
String^ path1 = "file1.txt";
String^ path2 = "file1.tmp";
array<Byte, 1>^ striqoni =
                gcnew array<Byte, 1>(15) {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n'};
//      file1.txt ფაილის შექმნა
FileStream^ fs = File::Open(path1, FileMode:: Create);
fs->Write(striqoni, 0, striqoni->Length);
fs->Close();
//      file1.tmp  ფაილის წაშლა
File::Delete(path2);

//      file1.txt ფაილის გადაწერა file1.tmp ფაილში
File::Copy(path1, path2);
label1->Text = path1 + L"  ფაილი გადაიწერა  " + path2 + L"ფაილში";
fs->Close();
}
```

მოცემული პროგრამით ხდება File კლასის GetAttributes()ფუნქციასთან მუშაობის დემონსტრირება.

```
{
//      File კლასის GetAttributes() ფუნქციასთან მუშაობის დემონსტრირება
FileAttributes^ faili = File::GetAttributes("file1.txt");
label1->Text = FileAttributes::Hidden.ToString();
label2->Text = faili->ToString();
}
```

მოცემული პროგრამით ხდება FileInfo კლასის თვისებებთან მუშაობის დემონსტრირება.

```
{
//      FileInfo კლასის თვისებებთან მუშაობის დემონსტრირება
System::IO::FileInfo^ obieqti = gcnew System::IO::FileInfo("file1.txt");

label1->Text += obieqti->Directory->ToString() + "\n";
label1->Text += obieqti->DirectoryName + "\n";
label1->Text += obieqti->FullName + "\n";
label1->Text += obieqti->Name + "\n";
label1->Text += obieqti->Extension + "\n";
label1->Text += obieqti->Length.ToString() + "\n";
```

```
label1->Text += obieqti->Exists.ToString() + "\n";
label1->Text += obieqti->CreationTime.ToString() + "\n";
label1->Text += obieqti->LastAccessTime.ToString() + "\n";
label1->Text += obieqti->LastWriteTime.ToString() + "\n";
}
```

კატალოგთან მუშაობა

System::IO სახელების სივრცეში განსაზღვრულია, აგრეთვე, Directory და DirectoryInfo კლასები. Directory კლასში განსაზღვრულია სტატიკური ფუნქციები (9.14 ცხრილი), DirectoryInfo კლასში კი - ჩვეულებრივი ფუნქციები (9.15 ცხრილი). რადგან Directory კლასის ფუნქციები სტატიკურია, ამიტომ მათთან სამუშაოდ არ არის საჭირო ობიექტის შექმნა, DirectoryInfo კლასის ფუნქციებთან სამუშაოდ კი საჭიროა ობიექტის შექმნა. 9.16 ცხრილში მოცემულია DirectoryInfo კლასის თვისებები.

მოვიყვანოთ რამდენიმე პროგრამა, რომლებშიც სრულდება კატალოგებთან მუშაობის დემონსტრირება Directory კლასის ფუნქციების გამოყენებით. ქვემოთ მოცემულ პროგრამაში ხდება ეკრანზე მითითებული კატალოგის სახელის, მასში მოთავსებული ფაილებისა და კატალოგების სახელების გაცემა, კატალოგის გახსნა, ეკრანზე დისკების სახელების გამოტანა. პროგრამაში კატალოგის სახელის შეტანა ხდება textBox1->Text კომპონენტში, მაგალითად, C:\\Windows. ამ დროს ბრჭყალების გამოყენება საჭირო არ არის.

ცხრილი 9.14. Directory კლასში განსაზღვრული ფუნქციები

ფუნქცია	აღწერა
DirectoryInfo^ CreateDirectory(String^ *კატალოგის_სახელი*)	ქმნის ახალ კატალოგს
void Delete(String^ *კატალოგის_სახელი*)	შლის კატალოგს მის შემცველობასთან ერთად
bool Exists(String^ *კატალოგის_სახელი*)	განსაზღვრავს არსებობს თუ არა კატალოგი
DateTime GetCreationTime(String^ *კატალოგის_სახელი*)	გასცემს კატალოგის შექმნის თარიღს
String^ GetCurrentDirectory()	გასცემს მიმდინარე კატალოგს
array<String^>^ GetDirectories(String^ *კატალოგის_სახელი*)	გასცემს მინდინარე კატალოგის ქვეკატალოგების სახელებს
String^ GetDirectoryRoot(String^ *კატალოგის_სახელი*)	მითითებული კატალოგისათვის გასცემს ძირითადი კატალოგის სახელს
array<String^>^ GetFiles(String^ *კატალოგის_სახელი*)	გასცემს მითითებულ კატალოგში მოთავსებული ყველა ფაილის სახელს
array<String^>^ GetFileSystemEntries(String^ *კატალოგის_სახელი*)	გასცემს მითითებულ კატალოგში მოთავსებული ყველა ქვეკატალოგისა და ფაილის სახელს
DateTime GetLastAccessTime(String^ *კატალოგის_სახელი*)	გასცემს კატალოგთან უკანასკნელი მიმართვის თარიღს
DateTime GetLastWriteTime(String^ *კატალოგის_სახელი*)	გასცემს კატალოგში უკანასკნელი ჩაწერის თარიღს

ცხრილი 9.14. (გაგრძელება)

array<String^>^ GetLogicalDrives()	გასცემს ლოგიკური დისკების სიას
DirectoryInfo^ GetParent(String^ *კატალოგის_სახელი*)	გასცემს მშობელი კატალოგის სახელს
void Move(String^ *საწყისი_კატალოგის_სახელი*, String^ *მიმღები_კატალოგის_სახელი*)	ახდენს კატალოგის გადატანას
void SetCreationTime(String^ *კატალოგის_სახელი*, DateTime::*შექმნის_დრო*)	აყენებს კატალოგის შექმნის თარიღს
void SetCurrentDirectory(String^ *კატალოგის_სახელი*)	აყენებს მიმდინარე კატალოგს
void SetLastAccessTime(String^ *კატალოგის_სახელი*, DateTime::*უკანასკნელი_მიმართვის_დრო*)	აყენებს კატალოგთან უკანასკნელი მიმართვის თარიღს
void SetLastWritetTime(String^ *კატალოგის_სახელი*, DateTime::*უკანასკნელი_ჩაწერის_დრო*)	აყენებს კატალოგში უკანასკნელი ჩაწერის თარიღს

ცხრილი 9.15. DirectoryInfo კლასში განსაზღვრული ფუნქციები

ფუნქცია	აღწერა
void Create()	ქმნის ახალ კატალოგს
DirectoryInfo^ CreateSubdirectory(String^ *კატალოგის_სახელი*)	ქმნის ახალ ქვეკატალოგს
void Delete()	შლის კატალოგს
array<DirectoryInfo^>^ GetDirectories()	გასცემს მიმდინარე კატალოგის ქვეკატალოგების სიას
array<FileInfo^>^ GetFiles()	გასცემს მიმდინარე კატალოგის ფაილების სიას
void MoveTo(String^ *მიმღები_კატალოგის_სახელი*)	ახდენს კატალოგის გადატანას

ცხრილი 9.16. DirectoryInfo კლასში განსაზღვრული თვისებები

ფუნქცია	ტიპი	აღწერა
Attributes	FileAttributes	გასცემს ან აყენებს კატალოგის ატრიბუტებს
CreationTime	DateTime	გასცემს ან აყენებს კატალოგის შექმნისას
Exist	bool	ამოწმებს კატალოგის არსებობას
Extension	String^	გასცემს კატალოგის გაფართოებას
FullName	String^	გასცემს გზას კატალოგისკენ და კატალოგის სახელს
LastAccessTime	DateTime	გასცემს ან აყენებს კატალოგთან უკანასკნელი მიმართვისას
LastWriteTime	DateTime	გასცემს ან აყენებს კატალოგში უკანასკნელი ჩაწერისას
Name	String^	გასცემს კატალოგის სახელს
Parent	String^	გასცემს მშობელი კატალოგის სახელს
Root	String^	გასცემს ძირითადი კატალოგის სახელს მითითებული კატალოგისათვის

```
{
//      მიმდინარე და მშობელი კატალოგების სახელების, აგრეთვე,
//      მითითებულ კატალოგში მოთავსებული ფაილებისა და კატალოგების სახელების
```

```
//      გამოტანა ეკრანზე
label1->Text = "";
//      მიმდინარე კატალოგის სახელის გაცემა
label1->Text = Directory::GetCurrentDirectory() + "\n";
//      მშობელი კატალოგის სახელის გაცემა.
//      textBox1->Text კომპონენტში შეგვაქვს კატალოგის სახელი, მაგალითად C:\Windows
label1->Text = Directory::GetParent(textBox1->Text)->ToString()+ "\n";
//      მითითებული კატალოგის გახსნა
Directory::SetCurrentDirectory(textBox1->Text);
//      ეკრანზე ფაილების გამოტანა
array<String^>^ striqonebis_masivi_1 = gcnew array<String^> (100);
striqonebis_masivi_1 = Directory::GetFiles(textBox1->Text);
for each ( String^ s in striqonebis_masivi_1 )
        label2->Text += s + "\n";
//      ეკრანზე კატალოგების გამოტანა
array<String^>^ striqonebis_masivi_2 = gcnew array<String^> (100);
striqonebis_masivi_2 = Directory::GetDirectories(textBox1->Text);
for each ( String^ s in striqonebis_masivi_2 )
        label1->Text += s + '\n';
//      ეკრანზე დისკების გამოტანა
array<String^>^ striqonebis_masivi_3 = gcnew array<String^> (100);
striqonebis_masivi_3 = Directory::GetLogicalDrives();
for each ( String^ s in striqonebis_masivi_3 )
        label1->Text += s + "\n";
}
```

ქვემოთ მოცემული პროგრამით სრულდება კატალოგის შექმნის, წაშლისა და გადატანის დემონსტრირება. Move() ფუნქციაში ჯერ უნდა მივუთითოთ გადასატანი კატალოგის სახელი, შემდეგ კი - იმ კატალოგის სახელი, რომელშიც გადატანა სრულდება. მიმღები კატალოგი არ უნდა არსებობდეს. ორივე კატალოგი ერთ დისკზეა.

```
{
//      კატალოგის შექმნა, წაშლისა და გადატანის დემონსტრირება
label2->Text = "";
//      კატალოგის შექმნა
Directory::CreateDirectory(textBox1->Text);
//      კატალოგის წაშლა. წასაშლელი კატალოგი ცარიელი უნდა იყოს
Directory::Delete(textBox2->Text);
// კატალოგის გადატანა. Move ფუნქციაში ჯერ ეთითება საწყისი კატალოგი, შემდეგ კი - მიმღები.
//      მიმღები კატალოგი არ უნდა არსებობდეს
Directory::Move(textBox3->Text, "C:\\Katalogi2");
}
```

ქვემოთ მოცემული პროგრამით სრულდება კატალოგებთან მუშაობის დემონსტრირება DirectoryInfo კლასის ფუნქციების გამოყენებით. MoveTo() ფუნქცია ახდენს Katalogi3 კატალოგიდან ფაილებისა და კატალოგების გადატანას Katalogi4 კატალოგში. გადატანის შემდეგ Katalogi3 კატალოგი იშლება.

```
{
//      ეკრანზე მითითებული კატალოგის სახელისა და ატრიბუტების
```

```
//      გამოტანა, აგრეთვე, ერთი კატალოგის შემცველობის გადატანა მეორე კატალოგში
label1->Text = "";
DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\bb");
label1->Text += obj1->Name + "\n";
label1->Text += obj1->Attributes.ToString() + "\n";
//      Katalogi3 კატალოგის შემცველობის გადატანა Katalogi4 კატალოგში
obj1->MoveTo("C:\\Katalogi4");
}
```

ფაილის არჩევა

ამ თავში განხილულ პროგრამებთან მუშაობისას ფაილის სახელის მითითებისათვის საჭიროა პროგრამის კოდის შეცვლა, რაც ბუნებრივია, გარკვეულწილად ართულებს სხვადასხვა ფაილთან მუშაობას. ამ პრობლემის გადასაწყვეტად გამოიყენება OpenFileDialog კლასი, რომელიც განსაზღვრულია System.Windows.Forms სახელების სივრცეში. მისი ShowDialog() ფუნქცია გამოიყენება ფაილის გახსნის სტანდარტული დიალოგური ფანჯრის გამოსატანად ეკრანზე. ამ ფუნქციასთან მუშაობის დემონსტრირება ხდება ქვემოთ მოცემული პროგრამით.

```
{
//      Open ფანჯრის გახსნა
label1->Text = "";
//      დიალოგური ფანჯრის გახსნა
OpenFileDialog^ FailisGaxsna = gcnew OpenFileDialog();

if ( FailisGaxsna->ShowDialog() == System::Windows::Forms::DialogResult::OK )
{
        FileInfo^ faili = gcnew FileInfo(FailisGaxsna->FileName);
        label1->Text += faili->FullName + "\n";
        label1->Text += faili->Length.ToString() + "\n";
        label1->Text += faili->LastAccessTime.ToString() + "\n";
        label1->Text += faili->DirectoryName + "\n";
}
}
```

ქვემოთ მოცემული პროგრამით ხდება ფაილიდან მონაცემების წაკითხვის დემონსტრირება Open დიალოგური ფანჯრის მეშვეობით. FailisGaxsna ობიექტის ShowDialog() ფუნქცია ხსნის Open დიალოგურ ფანჯრას. მოვნიშნავთ საჭირო ფაილს (უმჯობესია .txt გაფართოების მქონე) და ვაჭერთ Open კლავიშს. მონიშნული ფაილის სახელი მიენიჭება FailisGaxsna ობიექტის FileName თვისებას.

```
{
//      ფაილიდან წაკითხვის დემონსტრირება Open დიალოგური
//      ფანჯრის საშუალებით
FileStream^ file_in;
String^ striqoni;
OpenFileDialog^ FailisGaxsna = gcnew OpenFileDialog();
if ( FailisGaxsna->ShowDialog() == System::Windows::Forms::DialogResult::OK )
{
        file_in = gcnew FileStream(FailisGaxsna->FileName, FileMode::Open );
```

```
        StreamReader^ file_stream_in = gcnew StreamReader(file_in);
        for ( ; ( striqoni = file_stream_in->ReadLine() )  != nullptr; )
                label1->Text += striqoni + "\n";
        file_stream_in->Close();
}
}
```

ქვემოთ მოცემული პროგრამით ხდება ფაილში მონაცემების ჩაწერის დემონსტრირება Save დიალოგური ფანჯრის გამოყენებით.

```
{
//      ფაილში ჩაწერის დემონსტრირება Save დიალოგური
//      ფანჯრის საშუალებით
String^ striqoni;
FileStream^ file_out;
SaveFileDialog^ FailisGaxsna = gcnew SaveFileDialog();
if ( FailisGaxsna->ShowDialog() == System::Windows::Forms::DialogResult::OK )
{
        file_out = gcnew FileStream(FailisGaxsna->FileName, FileMode::Create);
        StreamWriter^ file_stream_out = gcnew StreamWriter(file_out);
        striqoni = textBox1->Text;
        //      ფაილში striqoni სტრიქონის ჩაწერა
file_stream_out->Write(striqoni);
file_stream_out->Close();
}
}
```

NetworkStream კლასი

ეს კლასი გამოცხადებულია System.Net.Sockets სახელების სივრცეში. ის პროცესს ქსელის საშუალებით უგზავნის და მისგან იღებს ბაიტებს. პროცესი შესაძლებელია მუშაობდეს ქსელის ნებისმიერ კომპიუტერზე და იყოს ფაილი ან მონაცემების სხვა სათავსი.

ქსელური შეერთებები ეფუძნება *სოკეტებს*. სოკეტი შეიცავს ჰოსთის (კომპიუტერის) მისამართს და პორტის ნომერს. ჰოსთის მისამართი შეიძლება იყოს TCP/IP მისამართი ან ჰოსთის სახელი. რაც შეეხება პორტის ნომრებს, ისინი სამ ჯგუფად იყოფა:

– 0÷1023 პორტები დაკავებული აქვს ცნობილ სამსახურებს. მაგალითად, მე-80 პორტი გამოიყენება მონაცემების გადასაცემად HTTP პროტოკოლის საშუალებით.

– 1024÷49151 პორტები დაკავებული აქვს ნაკლებად ცნობილ სამსახურებს.

– 49152÷65536 პორტები განკუთვნილია მომხმარებლების მიერ შემუშავებული პროგრამა-დანართებისთვის და დინამიკური გამოყენებისთვის.

იმისათვის, რომ ორი პროგრამა ერთმანეთს დაუკავშირდეს ქსელის საშუალებით, თითოეულმა მათგანმა უნდა შექმნას სოკეტი, რომელსაც მიმართავს მეორე პროგრამა. ამ მიზნისათვის შეგვიძლია გამოვიყენოთ Socket კლასი, რომელიც განსაზღვრულია System::Net::Sockets სახელების სივრცეში. TopListener კლასი ხსნის სოკეტს და ელოდება მასთან კლიენტის მიერთებას. TopClient კლასი ხსნის სოკეტს და უერთდება სამსახურს დაშორებულ კომპიუტერზე.

მოცემული პროგრამით ხდება სერვერის რეალიზება. სერვერი ელოდება კლიენტის მიერთებას. კლიენტის მიერთების შემთხვევაში სერვერი მას გადასცემს ბუფერში მოთავსებულ

მონაცემებს.

```
//        სერვერის რეალიზება
ref class Serveri {
//        Listen  ფუნქცია ელოდება შეერთებას
public :
void Listen() {
System::Net::IPAddress^ Lokaluri_Host = System::Net::IPAddress::Parse("127.0.0.1");
//        50025 პორტის მოსმენა
System::Net::Sockets::TcpListener^ obj_tcp =
                                        gcnew System::Net::Sockets::TcpListener(Lokaluri_Host, 50025);
obj_tcp->Start();
//        კლიენტის შეერთების ლოდინი
for ( ; ; ) {
//        კლიენტის სოკეტთან შეერთება
System::Net::Sockets::Socket^ Axali_Socketi = obj_tcp->AcceptSocket();
if ( Axali_Socketi->Connected ) {
//        NetworkStream ობიექტის შექმნა სოკეტის გახსნისათვის
System::Net::Sockets::NetworkStream^ Qseluri_Nakadi = gcnew
System::Net::Sockets::NetworkStream(Axali_Socketi);
//        მონაცემების გადაგზავნა
array<System::Byte>^ buferi = gcnew
array<System::Byte> (4) { ( System::Byte ) 's', ( System::Byte ) 'a', ( System::Byte ) 'b', ( System::Byte ) 'a' };
Qseluri_Nakadi->Write(buferi, 0, 4);
Qseluri_Nakadi->Flush();
Qseluri_Nakadi->Close();
}
//        რესურსების გათავისუფლება
Axali_Socketi->Close();
break;
}
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
//        შესრულების ნაკადის გაშვება, რომელიც პორტს უსმენს
Serveri^ Serveri_1 = gcnew Serveri();
Serveri_1->Listen();
}
```

TcpListener კლასის Start() ფუნქცია იწყებს პორტის მოსმენას ქსელში ახალი შეერთების მოლოდინში, AcceptSocket() ფუნქცია კი ასრულებს კლიენტის რეალურ შეერთებას ანუ ქმნის ახალ სოკეტს შეერთებისათვის. შექმნილ სოკეტს გამოვიყენებთ კლიენტისათვის მონაცემების გადასაგზავნად. ეს პროგრამა ახდენს კლიენტის რეალიზებას და იგი უნდა გავაფორმოთ როგორც ახალი პროექტი.

```
//        კლიენტის რეალიზება
{
label1->Text = "";
//        კლიენტის სოკეტის შექმნა
```

```
System::Net::Sockets::TcpClient^ Klientis_Soketi = gcnew System::Net::Sockets::TcpClient("127.0.0.1",
50025);
//      NetworkStream ობიექტის შექმნა ჰოსთიდან მონაცემების წასაკითხად
System::Net::Sockets::NetworkStream^ Qseluri_Nakadi = Klientis_Soketi->GetStream();
//      ნაკადიდან ბაიტების მასივის წაკითხვა
array<System::Byte>^ buferi = gcnew array<System::Byte> (100);
Qseluri_Nakadi->Read(buferi, 0, 100);
//      ბაიტების გარდაქმნა სიმბოლოებად და ეკრანზე გამოტანა
array<System::Char>^ buferi2 = gcnew array<System::Char> (100);
for ( int i = 0 ; i < 100 ; i++ ) buferi2[i] = ( char ) buferi[i];
for ( int i = 0 ; i < 100 ; i++ ) label1->Text += buferi2[i].ToString();
//      ნაკადის დახურვა
Qseluri_Nakadi->Close();
}
```

როგორც ვხედავთ, TcpClient კლასის კონსტრუქტორს გადაეცემა ორი პარამეტრი: ჰოსთის მისამართი ან სახელი და პორტის ნომერი. კონსტრუქტორი ცდილობს დაუკავშირდეს შესაბამის სოკეტს ქსელში. შეერთების დამყარების შემდეგ ვიყენებთ GetStream() ფუნქციას, რომელიც გასცემს NetworkStream კლასის ობიექტს. ამის შემდეგ, ვიყენებთ NetworkStream კლასის Read() ფუნქციას სოკეტიდან მონაცემების მისაღებად.

ახლა, ჯერ გავუშვათ Serveri პროგრამა. ამისათვის, უნდა გავხსნათ ჩვენი პროექტის შემცველი კატალოგი, შემდეგ Debug ქვეკატალოგი და შევასრულოთ .exe გაფართოების მქონე პროგრამა. ის დაელოდება კლიენტის მხრიდან შეერთებას. შემდეგ, ასეთივე გზით გავუშვათ Clienti პროგრამა. კლიენტი მიუერთდება სერვერს და label1 კომპონენტში გამოჩნდება სტრიქონი „საბა". ამის შემდეგ, სერვერი და კლიენტი მუშაობას ამთავრებენ. თუ ჯერ გავუშვებთ კლიენტს და შემდეგ სერვერს, მაშინ გაიცემა შეტყობინება შეცდომის შესახებ (აღიძვრება SocketException განსაკუთრებული სიტუაცია), რადგან ვერ იქნება მოძებნილი სერვერი, რომელიც უსმენს 50025 პორტს.

MemoryStream და BufferedStream კლასები

MemoryStream და BufferedStream კლასების გამოყენება მონაცემების გადასაგზავნად ზრდის კოდის ეფექტურობას. საქმე ის არის, რომ ჩვენ იშვიათად გვიწევს იმ რაოდენობის მონაცემებთან მუშაობა, რომელსაც ოპერაციული სისტემა ეფექტურად დაამუშავებს. დავუშვათ, ფაილიდან გვინდა წავიკითხოთ 6 ბაიტი. ოპერაციული სისტემა ერთი ოპერაციის შედეგად რეალურად წაიკითხავს 4, 8 ან მეტ კილობაიტს. თუ გამოვიყენებთ BufferedStream კლასს, მაშინ ოპერაციული სისტემა მონაცემებს წარმოგვიდგენს ბლოკების სახით, რაც უზრუნველყოფს მაღალ მწარმოებლურობას. BufferedStream კლასი პროგრამას გადასცემს მხოლოდ იმ მონაცემებს, რომლებიც მოთხოვნილი იყო. გარდა ამისა, BufferedStream კლასი ოპერატიულ მეხსიერებაში შეიძლება ინახავდეს ჩაწერის რამდენიმე ოპერაციის შედეგს და ისინი მაშინ ჩაწეროს დისკზე, როდესაც ჩაწერა ყველაზე ეფექტური იქნება. ასეთი შიგა ბუფერირების გამო უნდა გამოვიყენოთ Flush() ფუნქცია BufferedStream ობიექტის შემცველობის რეალურად ჩასაწერად მასთან დაკავშირებული მონაცემების სათავსში.

MemoryStream კლასის ნაკადისათვის მონაცემების სათავსია ოპერატიული მეხსიერების უბანი. ამ პროგრამით ხდება პროგრამის ერთი ნაწილიდან მეორეში მონაცემების გადაცემის დემონსტრირება.

```
{
```

```
//      მონაცემების გადაცემის დემონსტრირება
//      MemoryStreamWrite ფუნქცია ინახავს MemoryStream ობიექტის
//      შემცველობას ფაილის სახით
static void MemoryStreamWrite(System::IO::MemoryStream^ Nakadi, String^ FailisSaxeli) {
FileStream^ Gamosasvleli_Nakadi = File::OpenWrite(FailisSaxeli);
Nakadi->WriteTo(Gamosasvleli_Nakadi);
Gamosasvleli_Nakadi->Flush();
Gamosasvleli_Nakadi->Close();
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
String^ failis_saxeli = "mexsiereba";
//      ფაილის წაკითხვა MemoryStream ობიექტში
FileStream^ Shesasvleli_Nakadi = File::OpenRead(failis_saxeli + ".txt");
System::IO::MemoryStream^ Mexsierebis_Nakadi = gcnew System::IO::MemoryStream();
//      მონაცემების გადაცემა სათავსში
Mexsierebis_Nakadi->SetLength(Shesasvleli_Nakadi->Length);
Shesasvleli_Nakadi->Read(Mexsierebis_Nakadi->GetBuffer(), 0, (int)Shesasvleli_Nakadi->Length);
//      ჩაწერის კორექტულად დამთავრება და რესურსების გათავისუფლება
Mexsierebis_Nakadi->Flush();
Shesasvleli_Nakadi->Close();
//      სათავსის გადაცემა ფუნქციისათვის ჩაწერის მიზნით
MemoryStreamWrite(Mexsierebis_Nakadi, failis_saxeli + ".bak");
Mexsierebis_Nakadi->Close();
}
```

MemoryStream კლასის ავტომატური კონსტრუქტორი ქმნის გაფართოებად ბუფერს, რომლის საწყისი ტევადობა ნულის ტოლია. ამ ბუფერის ზომა შეგვიძლია გავზარდოთ ამავე კლასის SetLength() ფუნქციის გამოყენებით. ბუფერთან პირდაპირი მიმართვისათვის გამოიყენება GetBuffer() ფუნქცია.

მონაცემების გადაგზავნა შესაძლებელია, აგრეთვე, BufferedStream კლასის გამოყენებით. მაგალითი:

```
//      მონაცემების გადაცემის დემონსტრირება
{
String^ failis_saxeli = "buferi";
//      საფაილო ნაკადების შექმნა
 FileStream^ Shesasvleli_Faili = File::OpenRead(failis_saxeli + ".txt");
 FileStream^ Gamosasvleli_Faili = File::OpenWrite(failis_saxeli + ".bak");
//      ბუფერირების დამატება
System::IO::BufferedStream^ Shemavali_Nakadi = gcnew System::IO::BufferedStream(Shesasvleli_Faili);
System::IO::BufferedStream^ Gamomavali_Nakadi = gcnew
System::IO::BufferedStream(Gamosasvleli_Faili);
array<System::Byte>^ buferi = gcnew array<System::Byte> (4096);
int WakitxuliBaitebisRaodenoba;
//      მონაცემების გადაწერა შესასვლელი ნაკადიდან გამოსასვლელ ნაკადში
//      ბუფერირების გამოყენებით
for ( ; WakitxuliBaitebisRaodenoba = Shemavali_Nakadi->Read(buferi, 0, 4096) > 0; )
Gamomavali_Nakadi->Write(buferi, 0, WakitxuliBaitebisRaodenoba);
```

```
//        გადაწერის კორექტულად დამთავრება და რესურსების გათავისუფლება
Gamomavali_Nakadi->Flush();
Gamomavali_Nakadi->Close();
Shemavali_Nakadi->Close();
Gamosasvleli_Faili->Close();
Shesasvleli_Faili->Close();
}
```

როგორც პროგრამიდან ჩანს, BufferedStream კლასის კონსტრუქტორს პარამეტრად გადაეცემა FileStream კლასის ობიექტი (ნაკადი). მიღებული ობიექტები გამოიყენება მონაცემების წასაკითხად და ჩასაწერად.

რაც შეეხება CryptoStream კლასს, მას მე-15 თავში განვიხილავთ.

მონაცემების ასინქრონული შეტანა-გამოტანა

ზემოთ განხილულ მაგალითებში ყველგან გამოიყენებოდა მონაცემების სინქრონული შეტანა-გამოტანა. ამ შემთხვევაში, პროგრამის შესრულება დროებით ჩერდება შეტანა-გამოტანის ოპერაციის დამთავრებამდე. ბუნებრივია, ეს იწვევს პროგრამის არასასურველ დაყოვნებასა და მოცდენას, განსაკუთრებით იმ შემთხვევებში, როდესაც პროგრამა კითხულობს ან წერს მონაცემების დიდ მასივს. ხშირ შემთხვევაში, პროგრამას შეუძლია შესრულების გაგრძელება მონაცემების შეტანა-გამოტანის ოპერაციის პარალელურად.

მონაცემების ასინქრონული შეტანა-გამოტანის შემთხვევაში შეტანა-გამოტანის ოპერაციების შესრულებისათვის შესრულების ცალკე ნაკადი იქმნება (შესრულების ნაკადებს მე-15 თავში განვიხილავთ). შეტანა-გამოტანის ოპერაციის დამთავრებისთანავე უკუგამოძახების ფუნქციის მიერ პროგრამა იღებს შესაბამის შეტყობინებას. უკუგამოძახების ფუნქცია გამოიძახება დელეგატის მიერ შეტანა-გამოტანის ოპერაციის დამთავრებისთანავე.

მოცემული პროგრამით ხდება ასინქრონული შეტანა-გამოტანის დემონსტრირება.

```
// მონაცემების ასინქრონული შეტანა-გამოტანის  დემონსტრირება
ref class Asinqronuli {
public :
//        წაკითხვის ნაკადი
static System::IO::FileStream^ ShesasvleliNakadi;
//        დელეგატი უკუგამოძახების ფუნქციის აღწერისათვის
static System::AsyncCallback^ Delegati;
//        დიდი ზომის ბუფერის გამოყოფა მონაცემების წაკითხვისათვის
static array<System::Byte>^ buferi = gcnew array<System::Byte> (500000);
//        უკუგამოძახების ფუნქცია, რომელიც გამოიძახება კითხვის დამთავრებისას
static void Damtavreba(System::IAsyncResult^ asyncResult) {
int bytesRead = ShesasvleliNakadi->EndRead(asyncResult);
System::Windows::Forms::MessageBox::Show(L"წაკითხულია " + bytesRead.ToString() + L" ბაიტი");
}
};
//
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e) {
String^ failis_saxeli = "asincronuli.txt";
//        ფაილის გახსნა
Asinqronuli::ShesasvleliNakadi = gcnew FileStream(failis_saxeli, FileMode::Open, FileAccess::Read,
```

```
FileShare::None, 2048, true);
//        დელეგატისთვის უკუგამოძახების ფუნქციის მინიჭება
Asinqronuli::Delegati = gcnew AsyncCallback(Asinqronuli::Damtavreba);
//        ასინქრონული წაკითხვა
Asinqronuli::ShesasvleliNakadi->BeginRead(Asinqronuli::buferi, 0, 500000, Asinqronuli::Delegati,
nullptr);
//        პარალელური გამოთვლები
for (int i = 0; i < 500; i++)
        label1->Text += i.ToString() + "  ";
}
```

როგორც პროგრამიდან ჩანს, გამოყენებულია FileStream() კონსტრუქტორის გადატვირთული ვერსია, რომელსაც ექვსი პარამეტრი აქვს. პირველი სამი პარამეტრის დანიშნულება ჩვენთვის ცნობილია. მეოთხე პარამეტრია FileShare, რომელიც მიუთითებს ფაილთან მუშაობა შესრულდება ერთობლივი თუ ექსკლუზიური გამოყენების რეჟიმში. მეხუთე პარამეტრი განსაზღვრავს შიგა ბუფერის ზომას. მეექვსე პარამეტრი მიუთითებს გახსნილი იქნება თუ არა ფაილი ასინქრონულ რეჟიმში სამუშაოდ. იმისათვის, რომ შევამოწმოთ ოპერაციული სისტემა უზრუნველყოფს თუ არა ასინქრონულ შეტანა-გამოტანას უნდა შევამოწმოთ FileStream ობიექტის IsAsync თვისება:

```
if ( Asinqronuli::ShesasvleliNakadi->IsAsync == true )
        label2->Text = L"სისტემა უზრუნველყოფს ასინქრონულ შეტანა-გამოტანას";
   else label2->Text = L"სისტემა არ უზრუნველყოფს ასინქრონულ შეტანა-გამოტანას";
```

პროგრამაში ფაილის გახსნის შემდეგ იქმნება Delegati დელეგატი, რომელიც შეტანა-გამოტანის ოპერაციის დამთავრების შემდეგ გამოიძახებს Damtavreba() ფუნქციას. შემდეგ იწყება მონაცემების ასინქრონული კითხვა, რომელსაც BeginRead() ფუნქცია ასრულებს. ეს ფუნქცია ქმნის შესრულების ახალ ნაკადს, რომელშიც შესრულდება კითხვის ოპერაცია. კითხვის დამთავრების შემდეგ BeginRead() ფუნქცია გამოიძახებს უკუგამოძახების Damtavreba() ფუნქციას. ამ ფუნქციის კოდში გამოყენებული EndRead() ფუნქცია გასცემს წაკითხული ბაიტების რაოდენობას. BeginRead() ფუნქციის მუშაობის პარალელურად სრულდება მის შემდეგ მოთავსებული კოდი, კერძოდ კი for ციკლი.

ბოლოს, შევნიშნოთ, რომ თუ ფაილი გახსნილია სინქრონული მიმართვისათვის, მაშინ ასინქრონული შეტანა-გამოტანის ფუნქციები მასთან მაინც სინქრონულ რეჟიმში იმუშავებს.

თავი 11. განსაკუთრებული სიტუაციები

განსაკუთრებული სიტუაციების დამუშავების საფუძვლები

განსაკუთრებულია სიტუაცია (exception), რომელიც გამოწვეულია პროგრამის შესრულებისას აღძრული შეცდომით. ასეთი სიტუაციებია: ნულზე გაყოფა, გადავსება, მასივის ინდექსის გასვლა დიაპაზონის გარეთ, არარსებული ფაილის გახსნის მცდელობა და ა.შ. ეს შეცდომები იწვევს პროგრამის შესრულების ავარიულად დამთავრებას. C++ ენაში არსებობს განსაკუთრებული სიტუაციის დამუშავების (exception handling) საშუალებები, რომელთა გამოყენებით შესაძლებელია ასეთი შეცდომის დამუშავება ისე, რომ არ შეფერხდეს პროგრამის შესრულება.

C++ ენაში განსაკუთრებული სიტუაცია წარმოდგენილია კლასებით. განსაკუთრებული სიტუაციების ყველა კლასი მემკვიდრეობით არის მიღებული განსაკუთრებული სიტუაციის Exception კლასისაგან, რომელიც System სახელების სივრცის ნაწილია. C++ ენაში განსაზღვრულია განსაკუთრებული სიტუაციების ორი კატეგორია: გენერირებული CLR გარემოს მიერ და გენერირებული პროგრამა-დანართების (Application) მიერ. პირველი მათგანი აღწერილია System.Exception კლაში, მეორე კი - System.ApplicationException კლასში.

ცხრილი 10.1. ხშირად აღძრული განსაკუთრებული სიტუაციები

განსაკუთრებული სიტუაცია	მნიშვნელობა
ArrayTypeMismatchException	მნიშვნელობის ტიპი შეუთავსებელია მასივის ტიპთან
Divide ByZeroException	ნულზე გაყოფის შეცდომა
IndexOutOfRangeException	მასივის ინდექსი გადის დიაპაზონის გარეთ
InvalidCastException	არაკორექტული გარდაქმნა პროგრამის შესრულების პროცესში
OutOfMemoryException	new ოპერატორის გამოძახება იყო არაკო-რექტული მეხსიერების უკმარისობის გამო
OverflowException	გადავსება არითმეტიკული ოპერაციის შესრულებისას
StackOverflowException	სტეკის გადავსება

ცხრილი 10.2. Exception კლასის თვისებები

თვისება	ტიპი	მნიშვნელობა
HelpLink	String^	ამ თვისებას შეიძლება მივანიჭოთ ფაილის სახელი ან web-მისამართი, რომელიც შეიცავს დამატებით ინფორმაციას განსაკუთრებული სიტუაციის შესახებ
Message	String^	შეიცავს განსაკუთრებული სიტუაციის აღწერას
Source	String^	შეიცავს განსაკუთრებული სიტუაციის გამომწვევი პროგრამის სახელს
StackTrace	String^	შეიცავს განსაკუთრებული სიტუაციის გამომწვევი ფუნქციისა და კლასის სახელს
TargetSite	MethodBase^	შიცავს იმ ფუნქციის სახელს, რომლიდანაც იყო გამოძახებული განსაკუთრებული სიტუაცია

System სახელების სივრცეში განსაზღვრულია ბევრი სტანდარტული ჩადგმული განსაკუთრებული სიტუაცია. 10.1 ცხრილში მოცემულია ხშირად აღძრული განსაკუთრებული სიტუაციები. Exception კლასის თვისებები მოცემულია 10.2 ცხრილში.

განსაკუთრებული სიტუაციების დამუშავება სრულდება სამი საკვანძო სიტყვის გამოყენებით: try, catch და throw. ისინი, ხშირ შემთხვევაში, ერთობლივად გამოიყენება. try ბლოკში მოთავსებულია პროგრამის ის ოპერატორები, რომელთა შესრულებასაც თვალყური უნდა ვადევნოთ. გენერირებულ განსაკუთრებულ სიტუაციას იჭერს და ამუშავებს catch ბლოკი. throw სიტყვა იწვევს განსაკუთრებული სიტუაციის ხელოვნურად გენერირებას.

try და catch ბლოკები

განსაკუთრებული სიტუაციის დამუშავება ეფუძნება try და catch ბლოკების გამოყენებას. ეს ბლოკები ერთობლივად მუშაობენ. მათი სინტაქსია:

```
try {
//      კოდი, რომლისთვისაც სრულდება შეცდომების მონიტორინგი
}
catch ( ტიპი_1 ობიექტი ) {
//      ExcepType1 განსაკუთრებული სიტუაციის დამუშავება
}
catch ( ტიპი_2 ობიექტი ) {
//      ExcepType2 განსაკუთრებული სიტუაციის დამუშავება
}       . . .
```

როგორც სინტაქსიდან ჩანს, შესამოწმებელი კოდი მოთავსებული უნდა იყოს ფიგურულ ფრჩხილებში. *ტიპი* პარამეტრი განსაზღვრავს დასამუშავებელი განსაკუთრებული სიტუაციის ტიპს. როდესაც catch ოპერატორი იჭერს განსაკუთრებული სიტუაციას, მაშინ შესაბამისი ობიექტი მიიღებს მის მნიშვნელობას. თუ განსაკუთრებული სიტუაციის დამამუშავებელი არ მიმართავს ობიექტს, რაც პრაქტიკულად ხშირად ხდება, მაშინ არ არის აუცილებელი მისი მითითება.

განსაკუთრებული სიტუაციის გენერირების შემდეგ მისი დაჭერა ხდება შესაბამისი catch ოპერატორით, რომელიც ახდენს ამ განსაკუთრებული სიტუაციის დამუშავებას. try ბლოკთან შეიძლება დაკავშირებული იყოს რამდენიმე catch ოპერატორი. განსაკუთრებული სიტუაციის ტიპი განსაზღვრულია catch ოპერატორში. თუ გენერირებული განსაკუთრებული სიტუაცია შეესაბამება catch ოპერატორში მითითებულ ტიპს, მაშინ ის დაიჭერს ამ განსაკუთრებულ სიტუაციას და შესრულდება ამ catch ოპერატორის კოდი. დანარჩენი catch ბლოკები იგნორირდება.

იმ შემთხვევაში, თუ განსაკუთრებული სიტუაციის გენერირება არ ხდება, მაშინ try ბლოკის ოპერატორების მუშაობა ჩვეულებრივი რიგითობით მთავრდება და მისი შესაბამისი catch ოპერატორები არ შესრულდება. პროგრამის შესრულება გაგრძელდება უკანასკნელი catch ოპერატორის შემდეგ მოთავსებული ოპერატორიდან. ამრიგად, catch ოპერატორები გამოიძახება მხოლოდ განსაკუთრებული სიტუაციების გენერირების შემთხვევაში.

როგორც ცნობილია, როდესაც ხდება ინდექსის გამოყენება, რომელიც გადის მასივის საზღვრებს გარეთ, აღიძვრება შეცდომა. ამ დროს გენერირდება განსაკუთრებული სიტუაცია. მოცემული პროგრამით ხდება ამ განსაკუთრებული სიტუაციის გენერირებისა და დაჭერის დემონსტრირება.

```
//      მასივის საზღვრებს გარეთ ინდექსის გასვლის შეცდომის დამუშავება
```

```
{
array <Int32>^ masivi = gcnew array<Int32>(5);
int indexi = 7;
try
{
label1->Text = L"ეს სტრიქონი გამოჩნდება განსაკუთრებული სიტუაციის გენერირებამდე";
//        აქ აღიძვრება განსაკუთრებული სიტუაცია
masivi[indexi] = 20;
label3->Text = L"ეს სტრიქონი არ გამოჩნდება";
}
//        განსაკუთრებული სიტუაციის დაჭერა
catch (IndexOutOfRangeException^ arg1)
{
label2->Text = L"ინდექსი დიაპაზონის გარეთაა " + arg1->ToString();
}
label4->Text = L"ეს ოპერატორი შესრულდება";
}
```

ამ პროგრამაში განსაკუთრებული სიტუაციის გენერირების შემთხვევაში მისი დაჭერა ხდება catch ოპერატორით. ამ მომენტიდან დაწყებული მუშაობას იწყებს catch ბლოკი, try ბლოკი კი მუშაობას ამთავრებს. შედეგად,

label3.Text = "ეს სტრიქონი არ გამოჩნდება";

ოპერატორი, რომელიც მოსდევს ინდექსის არასწორად გამოყენების ოპერატორს, არ შესრულდება. catch ბლოკის შესრულების შემდეგ მართვა გადაეცემა ოპერატორს, რომელიც მოსდევს ამ catch ბლოკს. ამრიგად, მიუხედავად აღძრული შეცდომისა, პროგრამა აგრძელებს მუშაობას და ავარიულად არ მთავრდება.

catch ოპერატორში შეიძლება არ იყოს მითითებული არც ერთი პარამეტრი. პარამეტრი მოითხოვება მხოლოდ მაშინ, როდესაც აუცილებელია განსაკუთრებული სიტუაციის ობიექტთან მიმართვა. რიგ შემთხვევებში განსაკუთრებული სიტუაციის ობიექტის მნიშვნელობა შეიძლება გამოყენებულ იქნეს განსაკუთრებული სიტუაციების დამამუშავებლის მიერ შეცდომის შესახებ დამატებითი ინფორმაციის მიღების მიზნით.

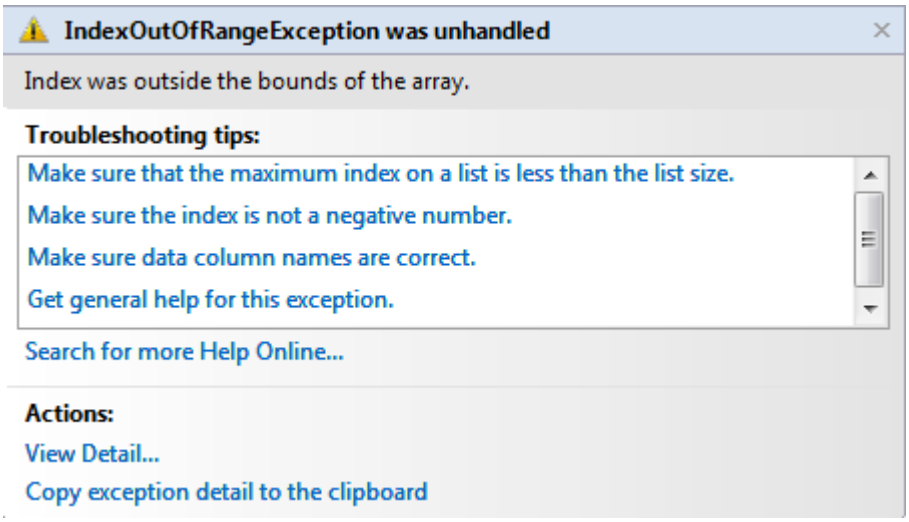
თუ არ მოხდა განსაკუთრებული სიტუაციის გენერირება, მაშინ catch ოპერატორი არ გამოიძახება და მართვა გადაეცემა catch ოპერატორის შემდეგ მყოფ ოპერატორს.

სტანდარტული განსაკუთრებული სიტუაციის დაჭერა და დამუშავება თავიდან აგვაცილებს პროგრამის ავარიულად დამთავრებას. თუ პროგრამა არ ახდენს განსაკუთრებული სიტუაციის დაჭერას, მაშინ განსაკუთრებული სიტუაციას დაიჭერს C++-ის შესრულების სისტემა. პრობლემა, რომელიც ამ შემთხვევაში ჩნდება ისაა, რომ შესრულების სისტემას ეკრანზე გამოაქვს შეტყობინება შეცდომის შესახებ და ავარიულად ამთავრებს პროგრამის შესრულებას. ქვემოთ მოცემული პროგრამით განსაკუთრებული სიტუაციის დაჭერა არ ხდება.

```
{
//        არ ხდება განსაკუთრებული სიტუაციის დაჭერა
array <Int32>^ masivi = gcnew array<Int32>(5);

masivi[0] = 5;    //    ეს ოპერატორი შესრულდება განსაკუთრებული სიტუაციის გენერირებამდე
masivi[6] = 20;   //    ინდექსის მნიშვნელობა გადის დიაპაზონის გარეთ
}
```

ამ შემთხვევაში პროგრამის შესრულება ავარიულად შეწყდება და გამოჩნდება შეტყობინება შეცდომის შესახებ (ნახ.10.1).



ნახ.10.1

მართალია, ეს შეტყობინება საკმაოდ სასარგებლოა პროგრამის გამართვისას, მაგრამ ის არ უნდა გამოჩნდეს პროგრამის ექსპლუატირებისას. ამიტომ, პროფესიულ დონეზე შედგენილმა პროგრამამ თვითონ უნდა დაამუშაოს განსაკუთრებული სიტუაციები.

როგორც ვიცით, განსაკუთრებული სიტუაციის ტიპი უნდა შეესაბამებოდეს catch ოპერატორში მითითებულ ტიპს. წინააღმდეგ შემთხვევაში განსაკუთრებული სიტუაცია არ იქნება დაჭერილი. ქვემოთ მოცემული პროგრამით ხდება ნულზე გაყოფით გამოწვეული განსაკუთრებული სიტუაციის (DivideByZeroException) დაჭერა. როგორც კი, ინდექსის მნიშვნელობა გავა დიაპაზონის გარეთ, მოხდება IndexOutOfRangeException განსაკუთრებული სიტუაციის გენერირება, რომელსაც ეს catch ოპერატორი ვერ იჭერს. შედეგად, პროგრამის შესრულება ავარიულად შეწყდება.

```
//      ხდება ნულზე გაყოფით გამოწვეული შეცდომის დაჭერა და
//      არ ხდება დიაპაზონის გარეთ ინდექსის გასვლის შეცდომის დაჭერა
{
array <Int32>^ masivi = gcnew array<Int32>(5);
int ricxvi;

try
{
label1->Text = "ეს სტრიქონი გამოჩნდება განსაკუთრებული სიტუაციის გენერირებამდე";
masivi[7] = 20;                 //      ინდექსის მნიშვნელობა გადის დიაპაზონის გარეთ
ricxvi = 10;                    //      ეს ოპერატორი არ შესრულდება
}
//      განსაკუთრებული სიტუაციის დაჭერა
catch ( DivideByZeroException^ )
{
label2->Text = "ინდექსი დიაპაზონის გარეთაა";
}
}
```

განსაკუთრებული სიტუაციების დამუშავების ერთ-ერთი მთავარი უპირატესობა ის არის, რომ პროგრამას შეუძლია მოახდინოს რეაგირება შეცდომაზე და შემდეგ გააგრძელოს შესრულება. ქვემოთ მოცემულ პროგრამაში ერთი მასივის ელემენტები იყოფა მეორე მასივის

ელემენტებზე. ნულზე გაყოფის შემთხვევაში გენერირდება DivideByZeroException განსაკუთრებული სიტუაცია. პროგრამა ახდენს მის დამუშავებას და გასცემს შესაბამის შეტყობინებას. ამრიგად, ნულზე გაყოფის შეცდომა არ გამოიწვევს პროგრამის ავარიულ გაჩერებას. ამის ნაცვლად, label კომპონენტში გამოჩნდება შესაბამისი შეტყობინება.

```
//      ნულზე გაყოფის შეცდომის დაჭერა მასივების გაყოფისას
{
array <Int32>^ masivi1 = gcnew array<Int32>{ 1, 3, 5, 7, 9 };
array <Int32>^ masivi2 = gcnew array<Int32>{ 0, 2, 4, 0, 3 };
array <Int32>^ masivi3 = gcnew array<Int32>(5);

label1->Text = "";
for ( int ind = 0; ind < masivi1.Length; ind++ )
try
{
masivi3[ind] = masivi1[ind] / masivi2[ind];
}
catch ( DivideByZeroException^ )
{
label2->Text += L"ადგილი აქვს ნულზე გაყოფას! \n";
}
}
```

ამ პროგრამაში try ბლოკი მოთავსებულია ციკლის ტანში, ამიტომ შესაძლებელი ხდება გამეორებადი შეცდომების დამუშავება.

try და catch ბლოკების გამოყენების მაგალითები

მოცემული პროგრამით სრულდება ფაილის გახსნისა და ფაილიდან წაკითხვის ოპერაციების შემოწმება. პროგრამის გაშვებამდე, ჩვენი პროექტის Debug კატალოგში (რომელშიც .exe გაფართოების ფაილია მოთავსებული) გადავწეროთ .txt გაფართოების მქონე ნებისმიერი ფაილი.

```
{
//      ფაილის გახსნისა და ფაილიდან წაკითხვის ოპერაციების მონიტორინგი
int ricxvi;
FileStream^ file1;
//      try  ბლოკი ამოწმებს ფაილის გახსნის ოპერაციას
try
{
        file1 = gcnew FileStream("filetext.txt", FileMode::Open);
}
catch (FileNotFoundException^ arg1)
{
label1->Text = arg1->Message;
return;
}
//      მონაცემების წაკითხვა ფაილიდან მანამ, სანამ არ შეგვხვდება EOF სიმბოლო
do
```

```
{
//        try  ბლოკი ამოწმებს ფაილიდან წაკითხვის ოპერაციას
try
{
ricxvi = file1->ReadByte();
}
catch ( IOException^ arg1 )
{
label2->Text = arg1->Message;
return;
}
if ( ricxvi != -1 ) label3->Text += ( Char ) ricxvi;
}
while ( ricxvi != -1 );                //        თუ ricxvi = -1, მაშინ მიღწეულია ფაილის დასასრული
file1->Close();
}
```

მოცემული პროგრამით სრულდება ფაილის შექმნისა და ფაილში ჩაწერის ოპერაციების შემოწმება. პროგრამაში გამოიყენება BinaryReader და BinaryWriter კლასები.

```
{
//        ფაილის შექმნისა და ფაილში ჩაწერის ოპერაციების მონიტორინგი
BinaryReader^ file_in;
BinaryWriter^ file_out;
int    ricxvii1 = 10, ricxvii2;
double wiladi1 = 1001.47, wiladi2, wiladi3;
bool   b1 = true, b2;
//        try  ბლოკი ამოწმებს ფაილის შექმნის ოპერაციას
try
{
        file_out = gcnew BinaryWriter(gcnew FileStream("file1.dat", FileMode::Create));
}
catch (IOException^ arg1)
{
        MessageBox::Show(arg1->Message + L"\n შეუძლებელია ფაილის შექმნა");
return;
}
//        try  ბლოკი ამოწმებს ფაილში ჩაწერის ოპერაციებს
try
{
//        ფაილში მთელი რიცხვის ჩაწერა
file_out->Write(ricxvii1);
//        ფაილში წილადის ჩაწერა
file_out->Write(wiladi1);
//        ფაილში ლოგიკური მონაცემის ჩაწერა
file_out->Write(b1);
//        ფაილში იწერება 10.2 * 2.3 გამოსახულების გამოთვლის შედეგი
file_out->Write(10.2 * 2.3);
}
catch (IOException^ arg1)
{
```

```
		MessageBox::Show(arg1->Message + L"\n ჩაწერის შეცდომა");
return;
}
file_out->Close();

//		ფაილიდან წაკითხვა
try
{
		file_in = gcnew BinaryReader(gcnew FileStream("file1.dat",FileMode::Open));
}
catch (IOException^ arg1)
{
		MessageBox::Show(arg1->Message + L"\n შეუძლებელია ფაილის გახსნა");
return;
}
try
{
//		მთელი რიცხვის წაკითხვა ფაილიდან
ricxvii2 = file_in->ReadInt32();
label1->Text = ricxvii2.ToString();
//		წილადის წაკითხვა ფაილიდან
wiladi2 = file_in->ReadDouble();
label2->Text = wiladi2.ToString();
//		ლოგიკური მონაცემის წაკითხვა ფაილიდან
b2 = file_in->ReadBoolean();
label3->Text = b2.ToString();
//		წილადის წაკითხვა ფაილიდან
wiladi3 = file_in->ReadDouble();
label4->Text = wiladi3.ToString();
}
catch (IOException^ arg1)
{
		MessageBox::Show(arg1->Message + L"\n წაკითხვის შეცდომა");
return;
}
file_in->Close();
}
```

მოცემული პროგრამით სრულდება ფაილის შექმნისა და ფაილში ჩაწერა-წაკითხვის ოპერაციების შემოწმება. გამოიყენება Seek().

```
{
//		პროგრამა 11.7
//		ფაილის შექმნისა და ფაილში ჩაწერა-წაკითხვის ოპერაციების მონიტორინგი
FileStream^ file;
//		try  ბლოკი ამოწმებს ფაილის შექმნის ოპერაციას
try
{
		file = gcnew FileStream("file.dat",FileMode::Create);
}
catch (FileNotFoundException^ arg1)
```

```
{
label2->Text = arg1->Message;
return;
}
Char simbolo;
label1->Text = "";
//        ასოების ჩაწერა ანბანის მიხედვით
for ( int i = 0; i < 26; i++ )
{
//        try  ბლოკი ამოწმებს ფაილში ჩაწერის ოპერაციას
try
{
file->WriteByte((Byte) ('a' + i));
}
catch (IOException^ arg1)
{
label2->Text = arg1->Message;
return;
}
}
//        try  ბლოკი ამოწმებს ფაილიდან წაკითხვის ოპერაციას
try
{
ile->Seek(0, SeekOrigin::Begin);                    //        პირველი ბაიტის არჩევა
simbolo = (Char) file->ReadByte();                  //        პირველი ბაიტის წაკითხვა
label1->Text += L"პირველი მნიშვნელობა    " + simbolo + "\n";
file->Seek(4, SeekOrigin::Begin);                   //        მეხუთე ბაიტის არჩევა
simbolo = (Char) file->ReadByte();                  //        მეხუთე ბაიტის წაკითხვა
label1->Text += L"მეხუთე მნიშვნელობა    " + simbolo + "\n";
}
catch (IOException^ arg1)
{
label3->Text = arg1->Message;
return;
}
file->Close();
}
```

რამდენიმე catch ბლოკის გამოყენება

ერთ try ბლოკთან შესაძლებელია დაკავშირებული იყოს რამდენიმე catch ბლოკი. მაგრამ, თითოეული catch ბლოკი იჭერს კონკრეტული ტიპის განსაკუთრებულ სიტუაციას.მოცემული პროგრამით ხდება ინდექსის მნიშვნელობის დიაპაზონის გარეთ გასვლისა და ნულზე გაყოფის შეცდომების დაჭერა.

```
//        რამდენიმე catch ოპერატორის გამოყენების დემონსტრირება
{
```

```
array <Int32>^ masivi1 = gcnew array<Int32>{ 1, 3, 5, 7, 9, 11, 13 };
array <Int32>^ masivi2 = gcnew array<Int32>{ 0, 2, 4, 0, 3 };
array <Int32>^ masivi3 = gcnew array<Int32>(7);

label1->Text = "";
for ( int ind = 0; ind < masivi1->Length; ind++ )
try
{
masivi3[ind] = masivi1[ind] / masivi2[ind];
}
catch ( DivideByZeroException^ )
{
label2->Text += L"ადგილი აქვს ნულზე გაყოფას! \n";
}
//      ეს catch ოპერატორი იჭერს მასივის ინდექსის დიაპაზონის გარეთ გასვლის შეცდომას
catch ( IndexOutOfRangeException^ )
{
label3->Text += L"ინდექსი დიაპაზონის გარეთაა! \n";
}
}
```

catch ოპერატორები მოწმდება იმ მიმდევრობით, როგორც პროგრამაშია მითითებული. შესრულდება ის ბლოკი, რომლის catch ოპერატორში მითითებული ტიპი დაემთხვევა განსაკუთრებული სიტუაციის ტიპს. დანარჩენი catch ბლოკები არ შესრულდება.

ყველა განსაკუთრებული სიტუაციის დაჭერა

ზოგჯერ საჭიროა ყველა განსაკუთრებული სიტუაციის დაჭერა მათი ტიპის მიუხედავად. ასეთ შემთხვევაში, catch ოპერატორში პარამეტრების ნაცვლად, უნდა მივუთითოთ სამი წერტილი. მოცემული პროგრამით ხდება IndexOutOfRangeException და DivideByZeroException განსაკუთრებული სიტუაციის დაჭერა და დამუშავების შესრულება ერთი catch ოპერატორით.

```
//      ყველა განსაკუთრებული სიტუაციის დაჭერა
{
array <Int32>^ masivi1 = gcnew array<Int32>{ 1, 3, 5, 7, 9, 11, 13 };
array <Int32>^ masivi2 = gcnew array<Int32>{ 0, 2, 4, 0, 3 };
array <Int32>^ masivi3 = gcnew array<Int32>(7);

label1->Text = "";
label2->Text = "";
for ( int ind = 0; ind < masivi1->Length; ind++ )
try
{
masivi3[ind] = masivi1[ind] / masivi2[ind];
}
//      ეს catch ოპერატორი ყველა შეცდომას იჭერს
```

```
catch (...)
{
label2->Text = L"გენერირდება განსაკუთრებული სიტუაცია!";
}
}
```

ჩადგმული try ბლოკები

შესაძლებელია try ბლოკების ერთმანეთში მოთავსება. შიგა try ბლოკში გენერირებული განსაკუთრებული სიტუაცია, რომელიც ვერ დაიჭირა ამ შიგა try ბლოკის შესაბამისმა catch ბლოკმა, ვრცელდება გარე try ბლოკზე. მოცემული პროგრამით გენერირებულ IndexOutOfRangeException განსაკუთრებულ სიტუაციას, რომელსაც ვერ იჭერს შიგა catch ბლოკი, დაიჭერს გარე catch ბლოკი.

```
//      ჩადგმული try ბლოკების გამოყენების დემონსტრირება
{
array <Int32>^ masivi1 = gcnew array<Int32>{ 1, 3, 5, 7, 9, 11, 13 };
array <Int32>^ masivi2 = gcnew array<Int32>{ 0, 2, 4, 0, 3 };
array <Int32>^ masivi3 = gcnew array<Int32>(7);
label1->Text = ""; label2->Text = "";
try
{
for ( int ind = 0; ind < masivi1->Length; ind++ )
{
try
{
masivi3[ind] = masivi1[ind] / masivi2[ind];
}
catch ( DivideByZeroException^ )
{
label2->Text = L"ადგილი აქვს ნულზე გაყოფას!";
}
}
}
catch ( IndexOutOfRangeException^ )
{
label3->Text = L"ინდექსი გადის დიაპაზონის გარეთ!";
}
}
```

პროგრამაში ნულზე გაყოფის შედეგად გენერირებული განსაკუთრებული სიტუაცია მუშავდება შიგა try ბლოკში. ამასთან, პროგრამა მუშაობას აგრძელებს. დიაპაზონის გარეთ ინდექსის გასვლისას შეცდომას ხელში იგდებს გარე try ბლოკის შესაბამისი catch ოპერატორი.

ჩვეულებრივ, გარე try ბლოკი გამოიყენება სერიოზული შეცდომების დასაჭერად, შიგა try ბლოკები კი - ნაკლებად სერიოზული შეცდომების დასამუშავებლად. შეგვიძლია, აგრეთვე, გამოვიყენოთ გარე try ბლოკი უპარამეტრებო catch ოპერატორით იმ შეცდომების დასაჭერად, რომლებიც არ მუშავდებიან შიგა try ბლოკში.

განსაკუთრებული სიტუაციის იძულებით გენერირება

წინა მაგალითებში ხდებოდა C++ სისტემის მიერ ავტომატურად გენერირებული განსაკუთრებული სიტუაციის დაჭერა. მაგრამ, განსაკუთრებული სიტუაციის გენერირება შეიძლება, აგრეთვე, throw ოპერატორის საშუალებით. მისი სინტაქსია:

throw ***განსაკუთრებული_სიტუაცია;***

აქ *განსაკუთრებული_სიტუაცია* არის throw ოპერატორის მიერ აღძრული განსაკუთრებული სიტუაცია. throw ოპერატორის მეშვეობით შესაძლებელია ნებისმიერი ტიპის განსაკუთრებული სიტუაციის გენერირება. მოცემული პროგრამით ხდება int ტიპის განსაკუთრებული სიტუაციის გენერირება throw ოპერატორის მიერ.

```
//          throw ოპერატორთან მუშაობის დემონსტრირება
{
int ricxvi1, ricxvi2, shedegi;
try
{
ricxvi1 = Convert::ToInt32(textBox1->Text);
ricxvi2 = Convert::ToInt32(textBox2->Text);
//          განსაკუთრებული სიტუაციის გენერირება
if ( ricxvi2 == 0 ) throw 0;
          else shedegi = ricxvi1 / ricxvi2;
}
catch ( int par )
{
label1->Text = L"განსაკუთრებული სიტუაცია დაჭერილია! \n გამყოფი ნულის ტოლია - " +
par.ToString(); // გაიცემა 25
}
//          try-catch ბლოკის დასასრული
label1->Text = shedegi.ToString();
}
```

განსაკუთრებული სიტუაციის გენერირება შეგვიძლია, აგრეთვე არა try ბლოკიდან, არამედ იმ ფუნქციიდან რომელიც try ბლოკიდან იყო გამოძახებული. ამის დემონსტრირება ხდება მოცემულ პროგრამაში.

```
ref class Klasi {
public :
void Funqcia(int par) {
if ( par < 0 ) throw par + 10;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj = gcnew Klasi();
int ricxvi = Convert::ToInt32(textBox1->Text);
try {
obj->Funqcia(ricxvi);
}
catch(int par) {
```

```
label1->Text = L"დაჭერილია შეცდომა - " + par.ToString();
}
}
```

try ბლოკი შეგვიძლია მოვათავსოთ ფუნქციის შიგნით:

```
ref class Klasi {
public : void Funqcia(int par) {
try {
if ( par < 0 ) throw par + 10;
}
catch(int par) {
System::Windows::Forms::MessageBox::Show(L"დაჭერილია შეცდომა - " + par.ToString());
}
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj = gcnew Klasi();
int ricxvi = Convert::ToInt32(textBox1->Text);
obj->Funqcia(ricxvi);
}
```

თუ გვინდა განმეორებით აღვძრათ განსაკუთრებული სიტუაცია, მაშინ throw ოპერატორი უნდა მივუთითოთ პარამეტრების გარეშე. ამ შემთხვევაში, მიმდინარე განსაკუთრებული სიტუაცია გადაეცემა გარე try/catch ბლოკს. განსაკუთრებული სიტუაციის განმეორებით აღძვრის აზრი ის არის, რომ ამ განსაკუთრებული სიტუაციის დამუშავების შესაძლებლობა მიეცეს რამდენიმე დამამუშავებელს. განსაკუთრებული სიტუაცია განმეორებით შეიძლება აღიძრას მხოლოდ catch ბლოკის შიგნით ან იმ ფუნქციის შიგნით, რომელიც ამ ბლოკიდან გამოიძახება. განმეორებით აღვძრული განსაკუთრებული სიტუაცია არ მუშავდება ამავე catch ოპერატორის მიერ, არამედ გადაეცემა ამ catch ოპერატორის მიმართ გარე catch ოპერატორს. მოცემული პროგრამით ხდება განსაკუთრებული სიტუაციის განმეორებით აღძვრის დემონსტრირება.

```
//          განსაკუთრებული სიტუაციის განმეორებით აღძვრის დემონსტრირება
ref class Klasi {
public :
void Funqcia2(int par) {
try {
if ( par < 0 ) throw par + 10;
}
catch(int) {
        System::Windows::Forms::MessageBox::Show(L"დაჭერილია შეცდომა ფუნქციაში");
throw;
}
}
};
//
private: System::Void button12_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj = gcnew Klasi();
int ricxvi = Convert::ToInt32(textBox1->Text);
try {
```

```
obj->Funqcia2(ricxvi);
}
catch (int) {
label1->Text += L"დაჭერილია შეცდომა მთავარ პროგრამაში";
}
}
```

გამართვა

.NET გარემოს შემადგენლობაში შედის პროგრამა გამმართველი. ის საშუალებას გვაძლევს პროგრამის კოდი შევასრულოთ ბიჯობრივ რეჟიმში და ვნახოთ პროგრამის ცვლადების მნიშვნელობები. ეს, თავის მხრივ, აადვილებს პროგრამის კოდში შეცდომების პოვნას.

მანამ, სანამ დავიწყებდეთ გამმართველთან მუშაობას, შევქმნათ ახალი პროექტი. ფორმაზე მოვათავსოთ ერთი button და ერთი label კომპონენტი. button კომპონენტს მივაბათ პროგრამა, რომელიც შეკრებს მასივის ელემენტებს:

```
{
int jami = 0;
array<int>^ masivi = gcnew array<int> { 3, 1, 8, -4, 1 };
for (int indexi = 0; indexi < masivi.Length; indexi++)
   jami += masivi[indexi];
label1->Text = jami.ToString();
}
```

წყვეტის წერტილების შექმნა

წყვეტის წერტილი არის პროგრამის კოდის ის ადგილი (სტრიქონი), სადაც წყდება პროგრამის შესრულება. ამ სტრიქონში მოთავსებული გამოსახულება (ოპერატორი ან ფუნქცია) არ შესრულდება და მართვა მომხმარებელს გადაეცემა. წყვეტის წერტილის შესაქმნელად საჭირო გამოსახულების გასწვრივ მარცხენა ველში უნდა მოვათავსოთ კურსორი და დავაჭიროთ თაგვის მარცხენა კლავიშს. გამოჩნდება მუქი წრე. წყვეტის წერტილის შესაქმნელად შეგვიძლია, აგრეთვე, კურსორი მოვათავსოთ საჭირო გამოსახულების შიგნით და დავაჭიროთ F9 კლავიშს. პროგრამის კოდში შეგვიძლია შევქმნათ წყვეტის რამდენიმე წერტილი. ჩვენი პროგრამის კოდში კურსორი მოვათავსოთ for ოპერატორის შემცველ სტრიქონში და დავაჭიროთ F9 კლავიშს. შეიქმნება წყვეტის წერტილი (ნახ. 10.2).

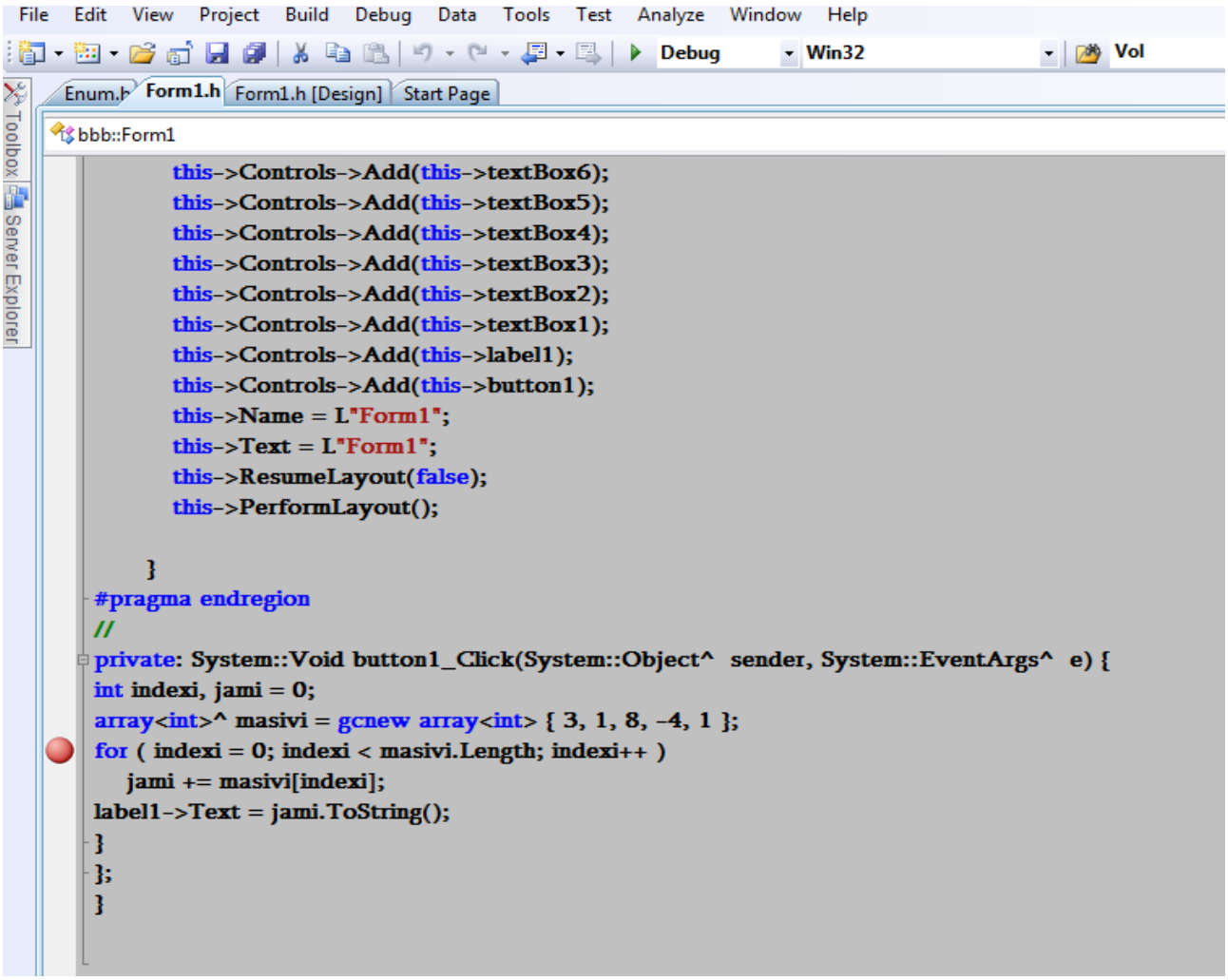
წყვეტის წერტილის გასაუქმებლად უნდა დავაჭიროთ მუქ წრეს ან F9 კლავიშს. წყვეტის წერტილთან პროგრამის კოდის შესრულება წყდება. ამის შემდეგ, პროგრამის კოდი შეგვიძლია ბიჯობრივ რეჟიმში შევასრულოთ და დავაკვირდეთ ცვლადების მნიშვნელობებს.

გამმართველის გაშვება ხდება F5 კლავიშით. თუ გვინდა კოდის შესრულება გამმართველის გარეშე, მაშინ უნდა დავაჭიროთ Ctrl+F5 კლავიშებს. დავაჭიროთ F5 კლავიშს. გამოჩნდება ფორმა. შემდეგ დავაჭიროთ button კლავიშს. გამმართველი კოდს წყვეტის წერტილამდე ასრულებს და მართვას გადასცემს მომხმარებელს. შესაბამისი გამოსახულების (int indexi = 0;) გასწვრივ მარცხენა ველში გამოჩნდება ისარი (ნახ. 10.3). ეს გამოსახულება ჯერ არ არის შესრულებული.

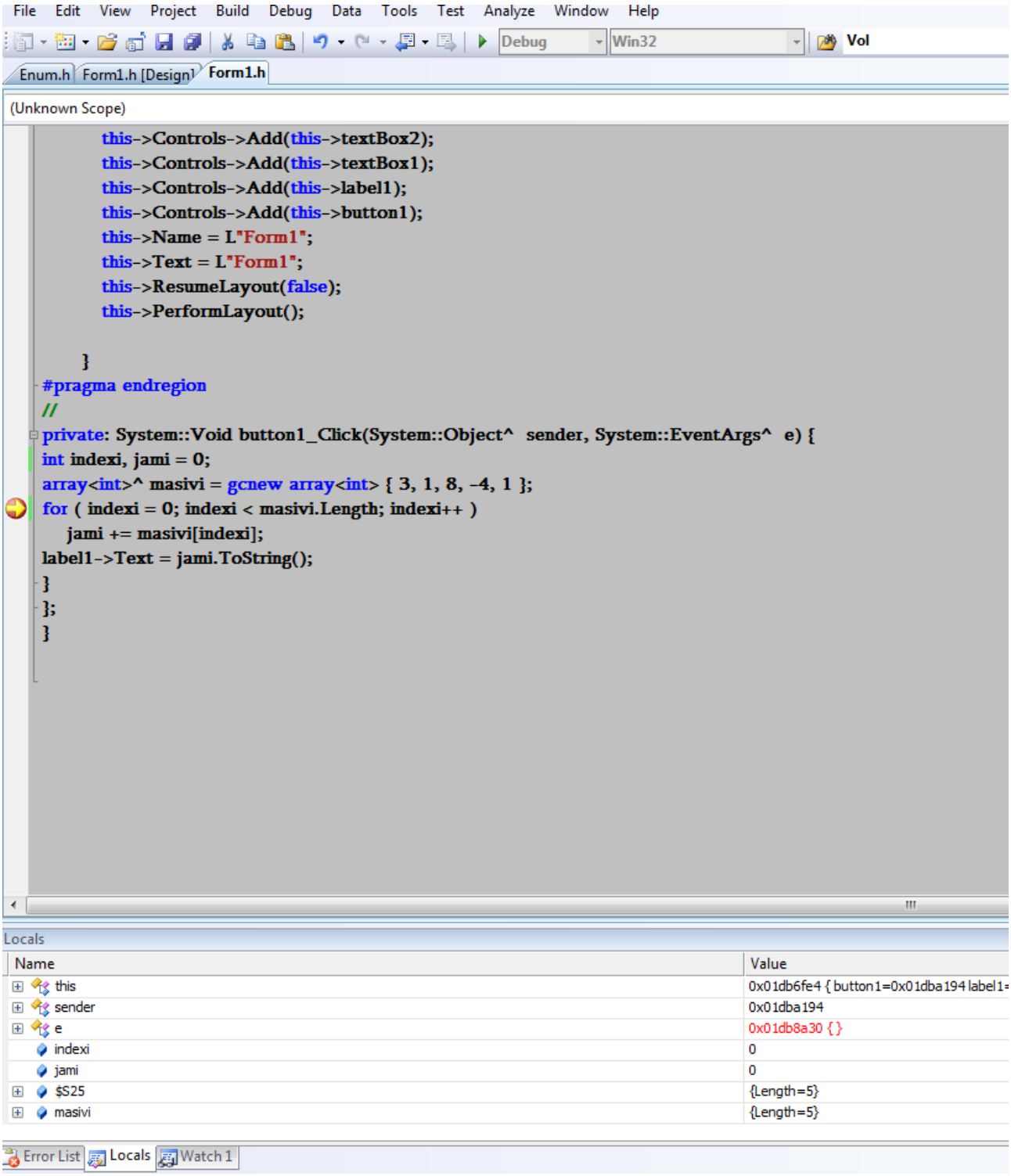
ცვლადების მნიშვნელობების ნახვა

როგორც აღვნიშნეთ, წყვეტის წერტილთან გამმართველი წყვეტს კოდის შესრულებას. ამ დროს შეგვიძლია ვნახოთ ცვლადების მნიშვნელობები.

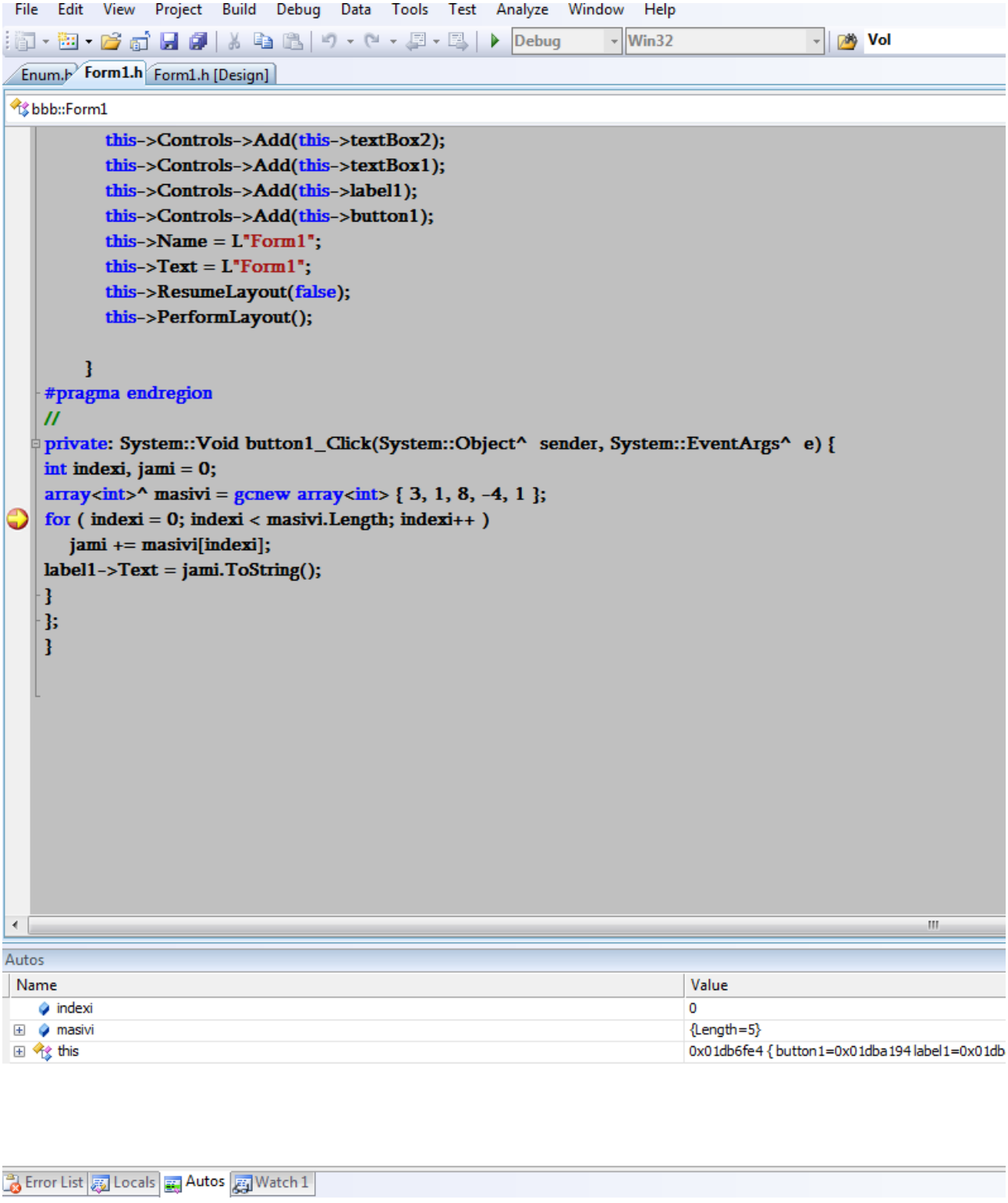
Autos ფანჯარა. ეს ფანჯარა მოთავსებულია გამმართველის ფანჯრის ქვემოთ (ნახ. 10.4). თუ ის არ ჩანს, მაშინ ვხსნით Debug მენიუს, Windows ქვემენიუს და ვასრულებთ Autos ბრძანებას (Debug→Windows→Autos). Windows ქვემენიუს ბრძანებები გამოჩნდება წყვეტის წერტილთან პროგრამის შეჩერების შემდეგ. ამ ფანჯარაში ჩანს იმ ცვლადების მნიშვნელობები, რომლებიც გამოიყენება მიმდინარე და წინა გამოსახულებებში. ამ ეტაპზე მასში ჩანს indexi, masivi და masivi.Length ცვლადების მნიშვნელობები. ცვლადის მნიშვნელობა შეგვიძლია ვნახოთ, აგრეთვე, თუ პროგრამის კოდში მის სახელზე გავაჩერებთ კურსორს. თუ Autos ფანჯარაში დავაჭერთ masivi სახელის მარცხნივ მოთავსებულ + ნიშანზე, მაშინ გამოჩნდება ამ მასივის ელემენტების მნიშვნელობები.



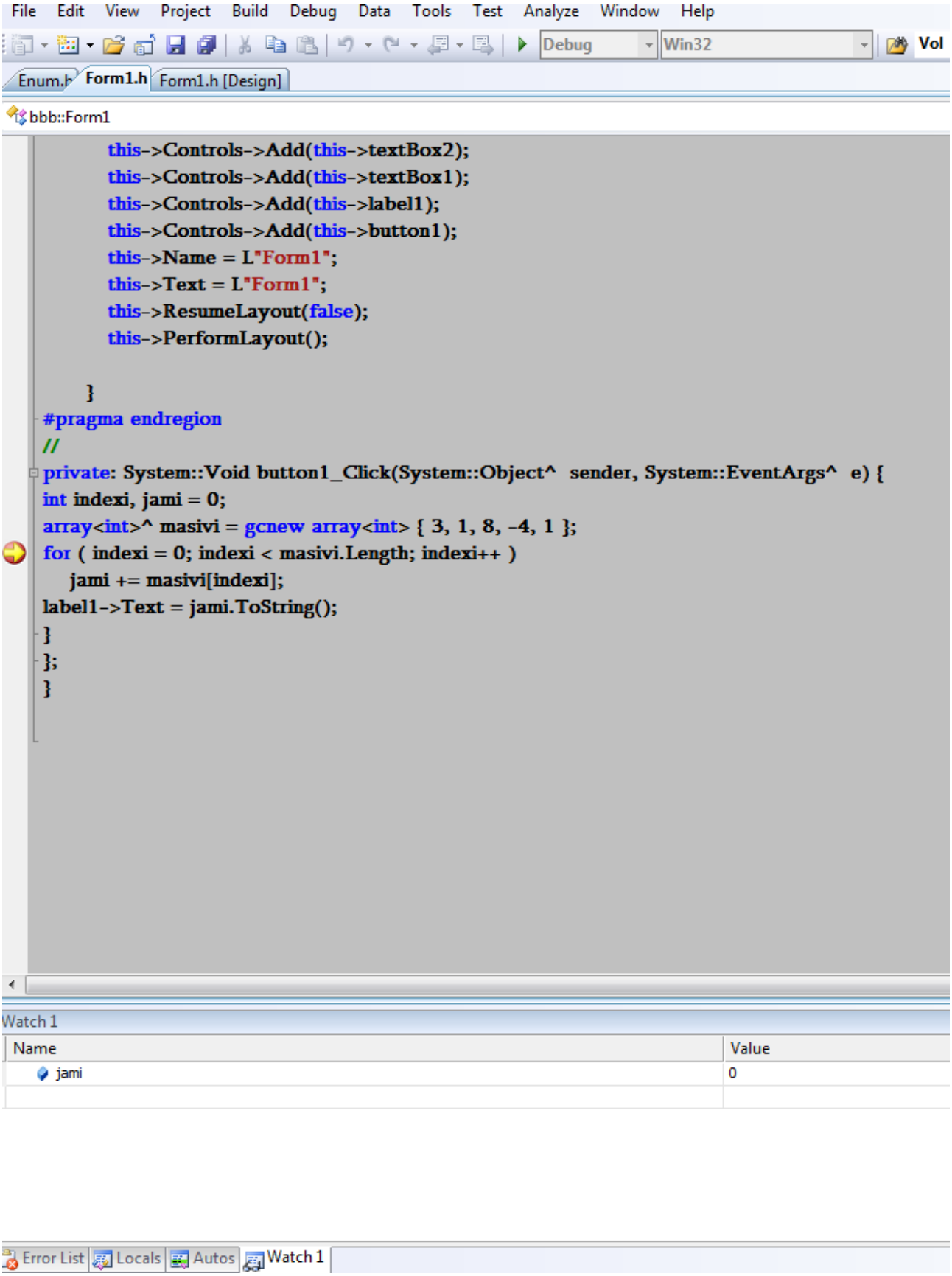
ნახ. 10.2.


File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
Enum.h
Form1.h [Design]
Form1.h
(Unknown Scope)
this->Controls->Add(this->textBox2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->label1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
this->PerformLayout();
}
#pragma endregion
//
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
int indexi, jami = 0;
array<int>^ masivi = gcnew array<int> { 3, 1, 8, -4, 1 };
for (indexi = 0; indexi < masivi.Length; indexi++)
jami += masivi[indexi];
label1->Text = jami.ToString();
}
};
}
Locals
Name
Value
this
0x01db6fe4 { button1=0x01dba194 label1=
sender
0x01dba194
e
0x01db8a30 { }
indexi
0
jami
0
$S25
{Length=5}
masivi
{Length=5}
Error List
Locals
Watch 1

Бзв. 10.3.


File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
Enum.h
Form1.h
Form1.h [Design]
bbb::Form1
this->Controls->Add(this->textBox2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->label1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
this->PerformLayout();
}
#pragma endregion
//
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
int indexi, jami = 0;
array<int>^ masivi = gcnew array<int> { 3, 1, 8, -4, 1 };
for (indexi = 0; indexi < masivi.Length; indexi++)
jami += masivi[indexi];
label1->Text = jami.ToString();
}
};
}
Autos
Name
Value
indexi
0
masivi
{Length=5}
this
0x01db6fe4 { button1=0x01dba194 label1=0x01db
Error List
Locals
Autos
Watch 1

Бъб. 10.4.


File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
Enum.h
Form1.h
Form1.h [Design]
bbb::Form1
this->Controls->Add(this->textBox2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->label1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
this->PerformLayout();
}
#pragma endregion
//
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
int indexi, jami = 0;
array<int>^ masivi = gcnew array<int> { 3, 1, 8, -4, 1 };
for (indexi = 0; indexi < masivi.Length; indexi++)
jami += masivi[indexi];
label1->Text = jami.ToString();
}
};
}
Watch 1
Name
Value
jami
0
Error List
Locals
Autos
Watch 1

Бљ. 10.5.

Locals ფანჯარა. ეს ფანჯარა მოთავსებულია გამმართველის ფანჯრის ქვემოთ (ნახ. 10.3). მის გასახსნელად უნდა დავაჭიროთ Locals ჩანართს. თუ ეს ფანჯარა არ ჩანს, მაშინ ვასრულებთ Debug→Windows→Locals ბრძანებას. მასში ჩანს იმ ცვლადების მნიშვნელობები, რომლებიც იმყოფებიან ხილვადობის მოცემულ უბანში. ამ მომენტისათვის ხილვადობის უბანში იმყოფება indexi, jami და masivi ცვლადები.

Watch ფანჯარა. ეს ფანჯარა მოთავსებულია გამმართველის ფანჯრის ქვემოთ (ნახ. 10.5). მის გასახსნელად უნდა დავაჭიროთ Watch ჩანართს. თუ ეს ფანჯარა არ ჩანს, მაშინ ვასრულებთ Debug→Windows→Watch→Watch1 ბრძანებას. მასში ჩანს სპეციალურად მითითებული ცვლადების მნიშვნელობები. ეს საჭიროა მაშინ, როდესაც ცვლადების დიდი რაოდენობიდან გვაინტერესებს მხოლოდ რამდენიმე. Watch ფანჯარაში ცვლადის დასამატებლად პროგრამის კოდში ცვლადის სახელზე უნდა დავაჭიროთ თაგვის მარჯვენა კლავიშს და კონტექსტური მენიუდან შევასრულოთ Add Watch ბრძანება. ასეთი გზით jami ცვლადი დავუმატოთ Watch ფანჯარას.

პროგრამის კოდის შესრულება ბიჯობრივ რეჟიმში

პროგრამის გამართვისას შეგვიძლია ცვლადების მნიშვნელობების ნახვა და ოპერატორების ბიჯობრივ რეჟიმში შესრულება. მიმდინარე გამოსახულების შესასრულებლად შეგვიძლია დავაჭიროთ F10 კლავიშს ან შევასრულოთ Debug→Step Over ბრძანება. თუ გვინდა ფუნქციის კოდის ბიჯობრივ რეჟიმში შესრულება, მაშინ უნდა დავაჭიროთ F11 კლავიშს ან შევასრულოთ Debug→Step Into ბრძანება. ფუნქციის კოდიდან გამოსასვლელად ვასრულებთ Debug→Step Out ბრძანებას ან ვაჭერთ Shift+F11 კლავიშებს. ფუნქციიდან გამოსვლისას მართვა გადაეცემა ამ ფუნქციის გამომძახებელ გამოსახულებას.

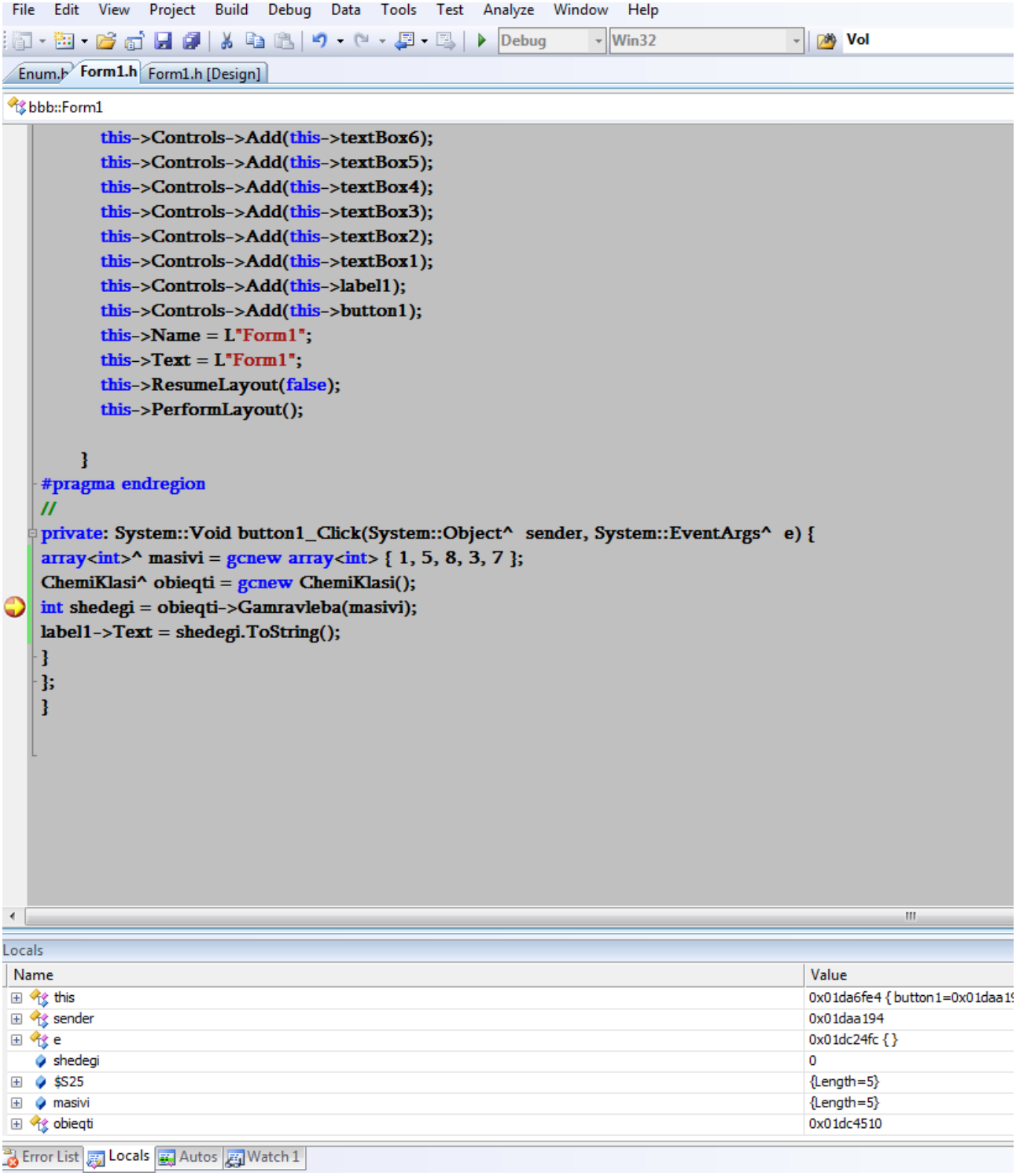
ახლა ჩვენ პროგრამას დავუმატოთ ChemiKlasi კლასი, რომელიც Gamravleba() ფუნქციას შეიცავს. ეს ფუნქცია ასრულებს მასივის ელემენტების ერთმანეთზე გამრავლებას და ნამრავლის გაცემას. კლასს შემდეგი სახე აქვს:

```
ref class ChemiKlasi {
public : int Gamravleba(array<int>^ mas1)
{
int namravli = 1;
for ( int indexi = 0; indexi < mas1->Length; indexi++ )
    namravli *= mas1[indexi];
return namravli;
}
};
```

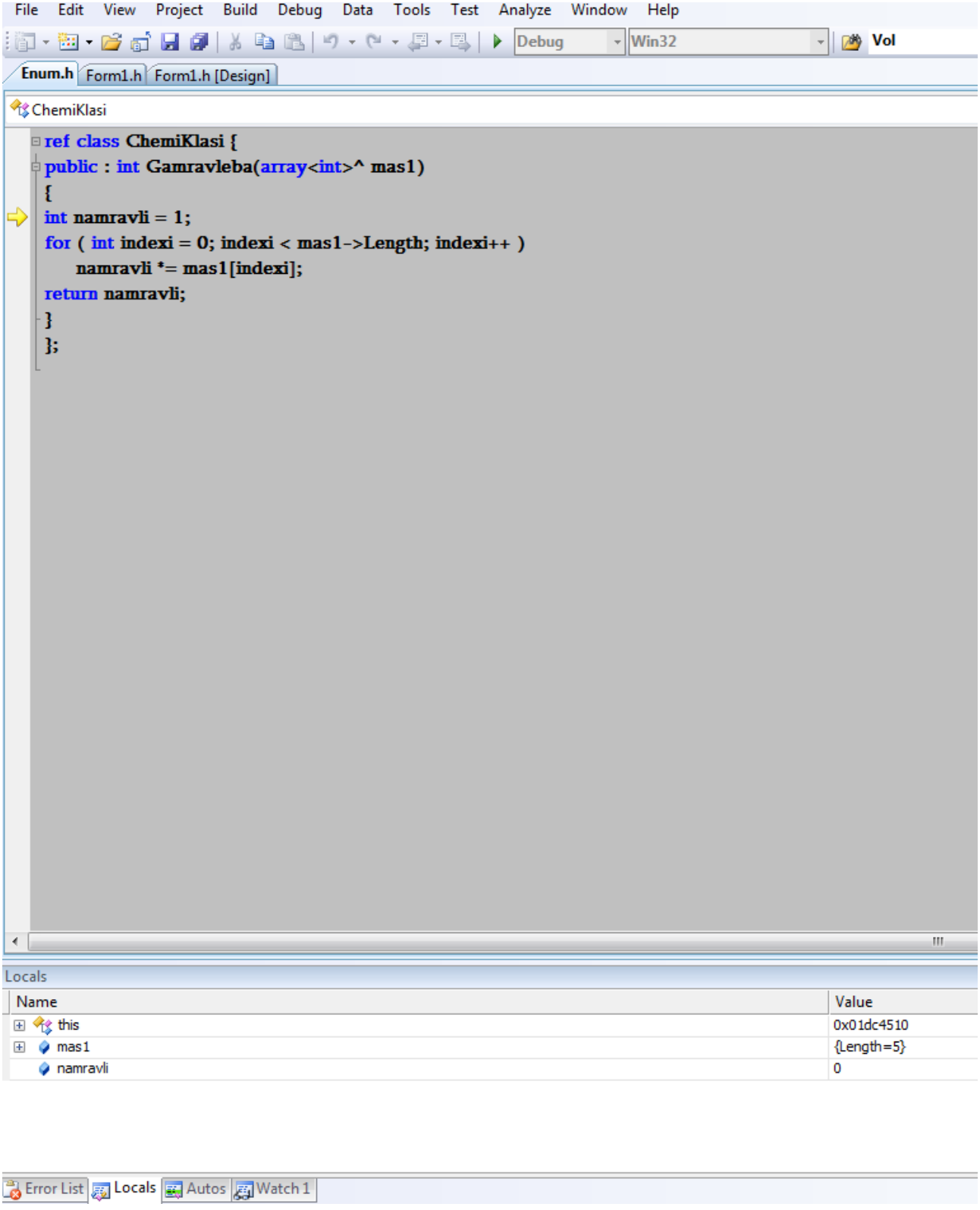
ფორმაზე მოვათავსოთ მეორე button კომპონენტი. მას მივაბათ პროგრამის კოდი, რომელიც Gamravleba() ფუნქციას გამოიძახებს:

```
{
array<int>^ masivi = gcnew array<int> { 1, 5, 8, 3, 7 };
ChemiKlasi^ obieqti = gcnew ChemiKlasi();
int shedegi = obieqti->Gamravleba(masivi);
label1->Text = shedegi.ToString();
}
```

ახლა სტრიქონში, რომელშიც ხდება Gamravleba() ფუნქციის გამოიძახება შევქმნათ წყვეტის წერტილი (ნახ. 10.6). დავაჭიროთ F5 კლავიშს. გაიხსნება ფორმა. დავაჭიროთ მეორე button კლავიშს. პროგრამის შესრულება შეწყდება წყვეტის წერტილში. ამის შემდეგ, თუ გვინდა Gamravleba() ფუნქციის შესრულება ბიჯობრივ რეჟიმში უნდა დავაჭიროთ F11 კლავიშს (ნახ. 10.7). ამ კლავიშზე ყოველი დაჭერის შემდეგ შეგვიძლია ვნახოთ ცვლადების მნიშვნელობები.


File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
Vol
Enum.h
Form1.h
Form1.h [Design]
bbb::Form1
this->Controls->Add(this->textBox6);
this->Controls->Add(this->textBox5);
this->Controls->Add(this->textBox4);
this->Controls->Add(this->textBox3);
this->Controls->Add(this->textBox2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->label1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
this->PerformLayout();
}
#pragma endregion
//
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
array<int>^ masivi = gcnew array<int> { 1, 5, 8, 3, 7 };
ChemiKlasi^ obieqti = gcnew ChemiKlasi();
int shedegi = obieqti->Gamravleba(masivi);
label1->Text = shedegi.ToString();
}
};
}
Locals
Name
Value
this
0x01da6fe4 { button1=0x01daa19
sender
0x01daa194
e
0x01dc24fc { }
shedegi
0
$S25
{Length=5}
masivi
{Length=5}
obieqti
0x01dc4510
Error List
Locals
Autos
Watch 1

ნახ. 10.6.


File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
Vol
Enum.h
Form1.h
Form1.h [Design]
ChemiKlasi
ref class ChemiKlasi {
public : int Gamravleba(array<int>^ mas1)
{
int namravli = 1;
for (int indexi = 0; indexi < mas1->Length; indexi++)
namravli *= mas1[indexi];
return namravli;
}
};
Locals
Name
Value
this
0x01dc4510
mas1
{Length=5}
namravli
0
Error List
Locals
Autos
Watch 1

ნახ. 10.7.

თავი 12. სტრიქონი

სტრიქონი

სტრიქონი გამოიყენება Unicode სიმბოლოების მიმდევრობის შესანახად. ერთი Unicode სიმბოლო იკავებს 2 ბაიტს ანუ 16 ბიტს. სტრიქონი System::String კლასის მიმართვითი ტიპის მქონე ობიექტია. ის აღიწერება String სიტყვის საშუალებით. სტრიქონის (String ტიპის ობიექტის) შექმნის ყველაზე მარტივი საშუალებაა სტრიქონული ლიტერალის გამოყენება. მაგალითად,

```
String^ striqoni = L"ეს არის სტრიქონული ლიტერალი";
```

აქ striqoni ცვლადი არის String ტიპის მიმართვითი ცვლადი, რომელსაც ენიჭება სტრიქონულ ლიტერალზე ("ეს არის სტრიქონული ლიტერალი") მიმართვა. ამ შემთხვევაში, striqoni ცვლადის ინიციალიზება ხდება სიმბოლოების მიმდევრობით - "ეს არის სტრიქონული ლიტერალი".

ისევე როგორც მასივში, აქაც პირველი ელემენტის (სიმბოლოს) ინდექსია 0. ინდექსი გამოიყენება მხოლოდ სიმბოლოს მისაღებად:

```
String^ striqoni = L"კომპიუტერი";
label1->Text = striqoni[2].ToString();
```

ორივე ოპერატორის შესრულების შედეგად label კომპონენტში გამოჩნდება ასო "მ".

striqoni ცვლადს მნიშვნელობა შეგვიძლია textBox კომპონენტიდანაც მივანიჭოთ:

```
String^ striqoni = textBox1->Text;
label1->Text = striqoni;
```

რადგან, სტრიქონი ობიექტია, ამიტომ მას Length თვისება აქვს. მასში ინახება სტრიქონის სიგრძე (სტრიქონში სიმბოლოების რაოდენობა):

```
{
String^ striqoni1 = L"რომან სამხარაძე";
String^ striqoni2 = L"საბა\nანა\n";
label1->Text = striqoni1->Length.ToString();          //      გაიცემა 9
label2->Text = striqoni2->Length.ToString();          //      გაიცემა 13
}
```

სტრიქონი შეიძლება შეიცავდეს მმართველ სიმბოლოებს:

```
String^ striqoni = L"საბა \n ანა";
label1->Text = striqoni;
```

ამ სტრიქონების შესრულების შედეგად label1 კომპონენტის პირველ სტრიქონში გამოჩნდება "საბა", მეორე სტრიქონში კი - "ანა".

ცხრილი 11.1. String კლასის თვისება

თვისება	ტიპი	აღწერა
Length	int	შეიცავს სტრიქონში სიმბოლოების რაოდენობას

ცხრილი 11.2. String კლასის ფუნქციები

ფუნქცია	აღწერა
static int Compare(String^ *სტრიქონი1*, String^ *სტრიქონი2*)	ადარებს ორ სტრიქონს. შედარებისას ითვალისწინებს ენისა და კულტურის თავისებურებებს
static int CompareOrdinal(String^ *სტრიქონი1*, String^ *სტრიქონი2*)	ადარებს ორ სტრიქონს. შედარებისას არ ითვალისწინებს ენისა და კულტურის თავისებურებებს
static String^ Concat(String^ *სტრიქონი1*, String^ *სტრიქონი2*)	გასცემს ორი ან მეტი სტრიქონის გაერთიანებას
static String^ Copy(String^ *სტრიქონი*)	გასცემს მითითებული სტრიქონის ასლს
static bool Equal(String^ *სტრიქონი1*, String^ *სტრიქონი2*)	გასცემს true მნიშვნელობას, თუ სტრიქონები ტოლია, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას
static String^ Format(String^ *სტრიქონი*, Object^ *ობიექტი*)	გასცემს მითითებული ფორმატის მიხედვით დაფორმატებულ სტრიქონს
static String^ Intern(String^ *სტრიქონი*)	გასცემს სისტემურ მიმართვას მითითებულ სტრიქონზე
static String^ IsInterned(String^ *სტრიქონი*)	გასცემს მიმართვას მითითებულ სტრიქონზე
static String^ Join(String^ *გამყოფი*, array<String^>^ *მასივი*)	ფუნქცია გამყოფს ათავსებს მასივის ელემენტებს შორის და გასცემს მიღებულ სტრიქონს
virtual int CompareTo(String^ *სტრიქონი*) sealed	გამომძახებელ სტრიქონს ადარებს მითითებულ სტრიქონთან
void CopyTo(int *საწყისი_ინდექსი*, array<Char>^ *სიმბოლოების_მასივი*, int *საბოლოო_ინდექსი*, int *რაოდენობა*)	გამომძახებელი სტრიქონის მითითებული პოზიციიდან სიმბოლოების მასივის მითითებულ პოზიციაში გადაწერს მითითებული რაოდენობის Unicode სიმბოლოს
bool EndsWith(String^ *სტრიქონი*)	ამოწმებს მთავრდება თუ არა გამომძახებელი სტრიქონი მითითებული სტრიქონით
Type^ GetType()	გასცემს მიმდინარე ეგზემპლარის ტიპს
int IndexOf(String^ *სტრიქონი*)	გასცემს გამომძახებელი სტრიქონის იმ პოზიციის ნომერს, სადაც პირველად იყო ნაპოვნი მითითებული ქვესტრიქონი
int IndexOfAny(array<Char>^ *მასივი*)	გასცემს გამომძახებელი სტრიქონის იმ პოზიციის ნომერს, სადაც პირველად იყო ნაპოვნი სიმბოლოების მასივის ნებისმიერი სიმბოლო
String^ Insert(int *ინდექსი*, String^ *სტრიქონი*)	გამომძახებელ სტრიქონში *ინდექსი* პოზიციიდან დაწყებული ჩასვამს მითითებულ სტრიქონს

ცხრილი 11.2. (გაგრძელება)

int LastIndexOf(String^ *სტრიქონი*)	გასცემს გამომძახებელი სტრიქონის იმ პოზიციის ნომერს, სადაც უკანასკნელად იყო ნაპოვნი მითითებული სტრიქონი
int LastIndexOfAny(array<Char>^ *მასივი*)	გასცემს გამომძახებელი სტრიქონის იმ პოზიციის ნომერს, სადაც უკანასკნელად იყო ნაპოვნი სიმბოლოების მასივის ნებისმიერი სიმბოლო
String^ PadLeft(int *სიგანე*, Char *სიმბოლო*)	სტრიქონის სიმბოლოებს ასწორებს მარცხენა საზღვარზე, რისთვისაც უმატებს სტრიქონის დასაწყისს ინტერვალებს ან მითითებულ სიმბოლოს იმდენჯერ, რომ სტრიქონის სიგრძე გახდეს მითითებული სიგანის ტოლი
String^ PadRight(int *სიგანე*, Char *სიმბოლო*)	სტრიქონის სიმბოლოებს ასწორებს მარჯვენა საზღვარზე, რისთვისაც უმატებს სტრიქონის ბოლოს ინტერვალებს ან მითითებულ სიმბოლოს იმდენჯერ, რომ სტრიქონის სიგრძე გახდეს მითითებული სიგანის ტოლი
String^ Remove(int *საწყისი_ინდექსი*, int *რაოდენობა*)	გამომძახებელ სტრიქონში *ინდექსი* პოზიციიდან დაწყებული წაშლის მითითებული რაოდენობის სიმბოლოს
String^ Replace(String^ *ძველი_სტრიქონი*, String^ *ახალი_სტრიქონი*)	გამომძახებელ სტრიქონში ძველ სტრიქონს ახალი სტრიქონით შეცვლის
array<String^>^ Split(array<Char>^ *გამყოფი*)	სტრიქონს ჰყოფს სტრიქონების მასივად მითითებული გამყოფის გამოყენებით.
bool StartsWith(String^ *სტრიქონი*)	ამოწმებს იწყება თუ არა გამომძახებელი სტრიქონი მითითებული სტრიქონით
String^ Substring(int *საწყისი_ინდექსი*, int *რაოდენობა*)	გამომძახებელი სტრიქონის ინდექსი პოზიციიდან გასცემს მითითებული რაოდენობის სიმბოლოს
array<Char>^ ToCharArray()	სტრიქონის სიმბოლოებს გადაწერს სიმბოლოების მასივში
String^ ToLower()	გამომძახებელი სტრიქონის ყველა სიმბოლო გადაჰყავს ქვედა რეგისტრში
virtual String^ ToString()	გამომძახებელ ობიექტს სტრიქონად გარდაქმნის
String^ ToUpper()	გამომძახებელი სტრიქონის ყველა სიმბოლო გადაჰყავს ზედა რეგისტრში
String^ Trim()	გამომძახებელ სტრიქონში შლის საწყის და ბოლო ინტერვალებს
String^ TrimEnd(array<Char>^ *სიმბოლოები*)	გამომძახებელი სტრიქონის ბოლოში შლის სიმბოლოების მასივში მითითებულ სიმბოლოებს
String^ TrimStart(array<Char>^ *სიმბოლოები*)	გამომძახებელი სტრიქონის დასაწყისში შლის სიმბოლოების მასივში მითითებულ სიმბოლოებს

სტრიქონებთან სამუშაო ფუნქციები

String კლასის თვისება და ფუნქციები მოცემულია 11.1 და 11.2 ცხრილებში. განვიხილოთ ზოგიერთი ფუნქცია.

ქვემოთ მოცემული პროგრამით ხდება **Insert, Remove** და **Replace** ფუნქციების გამოყენების დემონსტრირება.

```
{
//      Insert, Remove და Replace ფუნქციებთან მუშაობის დემონსტრირება
String^ str1;
String^ str2 = L"C++ დაპროგრამების ენაა";
String^ str3 = L"თანამედროვე ";
str1 = str2->Insert(17, str3);      //      მიიღება სტრიქონი "C++ დაპროგრამების თანამედროვე ენაა"
label1->Text = str1;
str1 = str2->Remove(3, 14);                         //      მიიღება სტრიქონი "C++ ენაა"
label2->Text = str1;
str1 = str2->Replace(L"დაპროგრამების ", str3);   //      მიიღება სტრიქონი "C++ თანამედროვე ენაა"
label3->Text = str1;
}
```

პროგრამის str1 = str2->Insert(17, str3); სტრიქონში str2 არის გამომძახებელი სტრიქონის სახელი. მის მარჯვნივ ისრის დასმის შემდეგ გაიხსნება სტრიქონებთან სამუშაო ფუნქციების სია. ვირჩევთ Insert ფუნქციას. მისი შესრულების შედეგად str2 სტრიქონის მე-17 პოზიციიდან მოხდება str3 სტრიქონის ჩასმა, ანუ "თანამედროვე" სტრიქონის ჩასმა. შედეგად, მიიღება სტრიქონი - "C++ დაპროგრამების თანამედროვე ენაა", რომელიც მიენიჭება str1 ცვლადს.

+ ოპერატორი გამოიყენება სტრიქონების კონკატენაციის (შეერთების) ოპერაციის შესასრულებლად. განვიხილოთ კოდის ფრაგმენტი:

```
{
String^ striqoni1 = L"ეს არის ";
String^ striqoni2 = L"კომ";
String^ striqoni3 = L"პიუტერი";
String^ striqoni   = striqoni1 + striqoni2 + striqoni3;
label1->Text = striqoni;
}
```

ამ ოპერატორების შესრულების შედეგად striqoni ცვლადს მიენიჭება სტრიქონი - "ეს არის კომპიუტერი".

ორი სტრიქონის შესადარებლად შეგვიძლია == ოპერატორის გამოყენება.

მოცემული პროგრამით ხდება **Copy, CompareTo, IndexOf, LastIndexOf, Substring** ფუნქციების მუშაობის დემონსტრირება.

```
{
//      Copy, CompareTo, IndexOf, LastIndexOf, Substring
//      ფუნქციებთან მუშაობის დემონსტრირება
String^ striqoni1;
String^ striqoni2 = L"ერთი ორი სამი ერთი ორი სამი";
String^ striqoni3 = textBox1->Text;
int shedegi, indexi;

//      striqoni2  სტრიქონი გადაიწერება striqoni1 სტრიქონში
striqoni1 = String::Copy(striqoni2);
label1->Text = striqoni1;
```

```
striqoni1 = striqoni2->Substring(9, 13);          //      მიიღება სტრიქონი "ერთი ორი სამი"
label2->Text = striqoni1;

indexi = striqoni2->IndexOf(L"ორი");              //      გაიცემა 5
label3->Text = indexi.ToString();
indexi = striqoni2->LastIndexOf(L"ორი");          //      გაიცემა 19
label4->Text = indexi.ToString();

//      striqoni2  და striqoni4 სტრიქონების შედარება
shedegi = striqoni2->CompareTo(striqoni3);
if ( shedegi == 0 ) label5->Text = L"სტრიქონები ერთმანეთის ტოლია";
        else if ( shedegi > 0 ) label5->Text = L" striqoni2 > striqoni3";
                        else label5->Text = L" striqoni2 < striqoni3";
}
```

Compare() და **Equals()** ფუნქციები. Equals() ფუნქცია ორ სტრიქონს ადარებს და გასცემს true მნიშვნელობას თუ სტრიქონები ტოლია, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას. მისი სინტაქსია:

static bool Equals(String^ *სტრიქონი1*, String^ *სტრიქონი2*)

მაგალითი:

```
{
String^ striqoni1 = L"საბა ანა ლიკა რომანი";
String^ striqoni2 = L"ანა რომანი საბა ლიკა";
bool shedegi = striqoni1->Equals(striqoni2);
if ( shedegi == true ) label1->Text = L"სტრიქონები ერთნაირია";
else label1->Text = L" სტრიქონები ერთნაირი არ არის";
}
```

Compare() ფუნქცია ორ სტრიქონს ადარებს ენის, ეროვნული და კულტურული თავისებურებების გათვალისწინებით. თუ პირველი სტრიქონი მეტია მეორეზე, მაშინ გაიცემა 1. თუ პირველი სტრიქონი ნაკლებია მეორეზე, მაშინ გაიცემა -1. თუ სტრიქონები ტოლია, მაშინ გაიცემა 0. ეს ფუნქცია გადატვირთულია და ამიტომ აქვს რამდენიმე ვერსია. ჩვენ განვიხილავთ რამდენიმე მათგანს.

Compare() ფუნქციის უმარტივესი ვერსიის სინტაქსია:

static int Compare(String^ *სტრიქონი1*, String^ *სტრიქონი2*)

მაგალითი:

```
{
String^ striqoni1 = L"საბა ანა ლიკა რომანი";
String^ striqoni2 = L"ანა რომანი საბა ლიკა";
int  shedegi = String::Compare(striqoni1, striqoni2);
if ( shedegi == 0 ) label1->Text = L"სტრიქონები ერთნაირია";
else label1->Text = L" სტრიქონები ერთნაირი არ არის";
}
```

Compare() ფუნქციის მეორე ვერსია შედარებისას ითვალისწინებს სიმბოლოების რეგისტრს. მისი სინტაქსია:

static int Compare(String^ *სტრიქონი1*, String^ *სტრიქონი2*, bool *რეგისტრი*)

თუ *რეგისტრი* იღებს true მნიშვნელობას, მაშინ შედარებისას სიმბოლოების რეგისტრი არ იქნება გათვალისწინებული. თუ ის იღებს false მნიშვნელობას, მაშინ შედარებისას

სიმბოლოების რეგისტრი გათვალისწინებული იქნება.

მაგალითი:

```
{
String^ striqoni1 = L"საბა ანა ლიკა რომანი";
String^ striqoni2 = L"ანა რომანი საბა ლიკა";
int  shedegi = String::Compare(striqoni1, striqoni2, true);
if ( shedegi == 0 ) label1->Text = L"სტრიქონები ერთნაირია";
else label1->Text = L" სტრიქონები ერთნაირი არ არის";
}
```

Compare() ფუნქციის მესამე ვერსია საშუალებას გვაძლევს შევადაროთ ორი სტრიქონის ნაწილები (ქვესტრიქონები). მისი სინტაქსია:

static int Compare(String^ *სტრიქონი1*, int *ინდექსი1*, String^ *სტრიქონი2*, int *ინდექსი2*,
int *სიმბოლოების_რაოდენობა*)

მოცემული პროგრამით ხდება ამ ფუნქციასთან მუშაობის დემონსტრირება.

```
{
String^ striqoni1 = L"საბა ანა ლიკა რომანი";
String^ striqoni2 = L"ანა რომანი საბა ლიკა";
int  shedegi = String::Compare(striqoni1, 5, striqoni2, 5, 4);
if ( shedegi == 0 ) label1->Text = L"სტრიქონები ერთნაირია";
else label1->Text = L" სტრიქონები ერთნაირი არ არის";
}
```

მოცემული პროგრამით ხდება **Concat()** ფუნქციასთან მუშაობის დემონსტრირება.

```
{
String^ striqoni1 = L"საბა";
String^ striqoni2 = String::Concat(striqoni1, L" ანა");
String^ striqoni3 = striqoni2 + L" ლიკა";
label1->Text = striqoni3;
}
```

როგორც პროგრამის კოდიდან ჩანს, სტრიქონების კონკატენაციისათვის შეიძლება გამოვიყენოთ + ოპერატორიც.

Format() ფუნქცია სტრიქონის დაფორმატებისათვის იყენებს დაფორმატების სიმბოლოებს, რომლებიც მოცემულია 11.3 ცხრილში.

ცხრილი 11.3. სტრიქონის დაფორმატების სიმბოლოები

დაფორმატების სიმბოლო	აღწერა
f ან F	რიცხვს აფორმატებს როგორც წილადს
e ან E	რიცხვს აფორმატებს როგორც რიცხვს ექსპონენციალური წარმოდგენით
p ან P	რიცხვს აფორმატებს როგორც პროცენტს
n ან N	რიცხვს აფორმატებს როგორც რიცხვს თანრიგების გამყოფებით
c ან C	რიცხვს აფორმატებს როგორც თანხას ადგილობრივ ვალუტაში
d ან D	რიცხვს აფორმატებს როგორც ათობით რიცხვს
g ან G	რიცხვს აფორმატებს როგორც წილადს ან როგორც რიცხვს ექსპონენციალური წარმოდგენით
x ან X	მთელი რიცხვი გადაჰყავს თექვსმეტობით წარმოდგენაში

მოცემული პროგრამით ხდება **Format()** ფუნქციის გამოყენების დემონსტრირება.

```
{
// Format() ფუნქციის გამოყენების დემონსტრირება
label1->Text = "";
String^ striqoni;
int mteli = 12;
double wiladi_d = 1234.56789;
float wiladi_f = 1234.56789f;
Decimal valuta = 1234567;
striqoni = String::Format("{0:D5}\n", mteli);                    //    00012
label1->Text += striqoni;
striqoni = String::Format("{0:X}\n", mteli);                     //    C
label1->Text += striqoni;
striqoni = String::Format("{0:P3}\n", mteli);                    //    1 200,000%
label1->Text += striqoni;
striqoni = String::Format("{0:C3}\n", valuta);                   //    1 234 567, 000 Lari
label1->Text += striqoni;
striqoni = String::Format("{0:F3}\n", wiladi_d);                 //    1234,568
label1->Text += striqoni;
striqoni = String::Format("{0:F3}\n", wiladi_f);                 //    1234,568
label1->Text += striqoni;
striqoni = String::Format("{0:P3}\n", wiladi_d);                 //    123 456,789%
label1->Text += striqoni;
striqoni = String::Format("{0:E3}\n", wiladi_d);                 //    1,235E+003
label1->Text += striqoni;
striqoni = String::Format("{0:N3}\n", wiladi_d);                 //    1 234,568
label1->Text += striqoni;
striqoni = String::Format("{0:G3}\n", wiladi_d);                 //    1,23E+03
label1->Text += striqoni;
}
```

{0:F3} ფორმატში პირველი სიმბოლო "0" მიუთითებს, რომ უნდა დაფორმატდეს რიგით პირველი ცვლადი. მეორე სიმბოლო მიუთითებს, რომ ათწილადში წილად ნაწილს უნდა დაეთმოს 3 თანრიგი. მოცემულ მაგალითში დაფორმატდება რიგით მეორე ცვლადი - wiladi_f, რადგან ფორმატში მითითებულია 1.

```
{
double wiladi_d = 9234.56781;
float wiladi_f = 1234.56789f;
String^ striqoni = String::Format("{1:F3}\n", wiladi_d, wiladi_f);          //    1234,568
label1->Text = striqoni;}
```

თუ ფორმატში არ მივუთითებთ წილადი ნაწილის თანრიგების რაოდენობას, მაშინ ის აიღება ორის ტოლი. მაგალითად,

```
{
float wiladi_f = 1234.56789f;
String^ striqoni = String::Format("{0:F}\n", wiladi_f);                     //    1234,56
label1->Text = striqoni;
}
```

მოცემული პროგრამით ხდება Join() ფუნქციასთან მუშაობის დემონსტრირება.

```
{
array<String^>^ striqonebis_masivi = { L"საბა", L"ანა", L"ლიკა", L"რომანი" };
String^ striqoni = String::Join("-", striqonebis_masivi);
label1->Text = striqoni;                    //      striqoni =  "საბა-ანა-ლიკა-რომანი"
}
```

მოცემული პროგრამით ხდება Split() ფუნქციასთან მუშაობის დემონსტრირება.

```
{
String^ striqoni = L"რომანი,ანა:საბა ლიკა.ნატა";
array<Char>^ simbolo = { ' ', ',', '.', ':' };
array<String^>^ shedegi;
shedegi = striqoni->Split(simbolo);

for each ( String^ sityva in shedegi )
{
if ( sityva->Trim() != L"" )
label1->Text += sityva + L'\n';          //      shedegi = "რომანი ანა საბა ლიკა ნატა"
}
}
```

მოცემული პროგრამით ხდება **StartWith()** და **EndWith()** ფუნქციასთან მუშაობის დემონსტრირება.

```
{
//      StartWith() და EndWith() ფუნქციებთან მუშაობის დემონსტრირება
String^  striqoni1;
String^ striqoni2;
striqoni1 = textBox1->Text;
striqoni2 = textBox2->Text;
if ( striqoni1->StartsWith(striqoni2) )
label1->Text = L"სტრიქონი იწყება " + striqoni2 + L" სტრიქონით";
        else  label1->Text = L" სტრიქონი არ იწყება " + striqoni2 + L" სტრიქონით";
if ( striqoni1->EndsWith(striqoni2) )
label2->Text = L"სტრიქონი მთავრდება " + striqoni2 + L" სტრიქონით";
        else  label2->Text = L" სტრიქონი არ მთავრდება " + striqoni2 + L" სტრიქონით";
}
```

IndexOfAny() და **LastIndexOfAny()** ფუნქციები გადატვირთულია და ამიტომ აქვთ რამდენიმე ვერსია. ყველაზე მარტივი ვერსიის სინტაქსია:

int IndexOfAny(array<Char>^ *სიმბოლოების_მასივი*)

int LastIndexOfAny(array<Char>^ *სიმბოლოების_მასივი*)

ამ ფუნქციების მეორე ვერსია საშუალებას გვაძლევს ძებნა დავიწყოთ მითითებული ინდექსდან. ამ ვერსიის სინტაქსია:

int IndexOfAny(array<Char>^ *სიმბოლოების_მასივი*, int *ინდექსი*)

int LastIndexOfAny(array<Char>^ *სიმბოლოების_მასივი*, int *ინდექსი*)

ამ ფუნქციების მესამე ვერსია საშუალებას გვაძლევს მივუთითოთ შესამოწმებელი სიმბოლოების რაოდენობა. ამ ვერსიის სინტაქსია:

int IndexOfAny(array<Char>^ *სიმბოლოების_მასივი*, int *საწყისი_ინდექსი*,
int *სიმბოლოების_რაოდენობა*)

int LastIndexOfAny(array<Char>^ *სიმბოლოების_მასივი*, int *საწყისი_ინდექსი*,
int *სიმბოლოების_რაოდენობა*)

მოცემული პროგრამით ხდება IndexOfAny() და LastIndexOfAny() ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      IndexOfAny() და LastIndexOfAny() ფუნქციებთან მუშაობის დემონსტრირება
array<Char>^ simboloebis_masivi = { L'ნ', L'ა' };
String^ striqoni1 = L"ლიკა, ანა და რომანი";

int indexi1 = striqoni1->IndexOfAny(simboloebis_masivi);                    //      indexi1 = 3
int indexi2 = striqoni1->LastIndexOfAny(simboloebis_masivi);                //      indexi2 = 17
label1->Text = indexi1.ToString() + "   " + indexi2.ToString();

int indexi3 = striqoni1->IndexOfAny(simboloebis_masivi, 5);                 //      indexi3 = 6
int indexi4 = striqoni1->LastIndexOfAny(simboloebis_masivi, 5);             //      indexi4 = 3
label2->Text += indexi3.ToString() + "   " + indexi4.ToString();

int indexi5 = striqoni1->IndexOfAny(simboloebis_masivi, 5, 6);              //      indexi5 = 6
int indexi6 = striqoni1->LastIndexOfAny(simboloebis_masivi, 5, 6);          //      indexi5 = 3
label3->Text += indexi5.ToString() + "   " + indexi6.ToString();
}
```

PadLeft() და **PadRight()** ფუნქციებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
String^ striqoni2 = L"საბა";
String^ striqoni1 = striqoni2->PadLeft(10, L'*');                           //      striqoni1 = "******საბა"
String^ striqoni3 = striqoni2->PadRight(10, L'*');                          //      striqoni3 = "საბა******"
label1->Text = striqoni1->ToString();
label2->Text = striqoni3->ToString();
}
```

Trim(), TrimStart() და **TrimEnd()** ფუნქციები გადატვირთულია. Trim() ფუნქციის ყველაზე მარტივი ვერსია, რომელშიც არ არის მითითებული პარამეტრი, შლის ინტერვალებს სტრიქონის დასაწყისსა და ბოლოში. მისი სინტაქსია:

String^ Trim()

ამ ფუნქციების მეორე ვერსია საშუალებას გვაძლევს სტრიქონს დასაწყისსა და ბოლოში მოვაცილოთ მითითებული სიმბოლოები. ამ ვერსიის სინტაქსია:

String^ Trim(array<Char>^ *სიმბოლოების_მასივი*)
String^ TrimStart(array<Char>^ *სიმბოლოების_მასივი*)
String^ TrimEnd(array<Char>^ *სიმბოლოების_მასივი*)

მოცემული პროგრამით ხდება Trim(), TrimStart() და TrimEnd() ფუნქციებთან მუშაობის დემონსტრირება.

```
{
String^ striqoni2;
String^ striqoni3;
String^ striqoni4 = L";,რომანი..;;";
array<Char>^ simboloebi = { L'.', L';', L',' };

label1->Text = striqoni4->Trim(simboloebi);
striqoni2 = striqoni4->TrimStart(simboloebi);                       //      striqoni2   = "რომანი..;;"
```

```
striqoni3 = striqoni4->TrimEnd(simboloebi);          //      striqoni3  = ";,რომანი"
label2->Text = striqoni2->ToString();
label3->Text = striqoni3->ToString();
}
```

მოცემული პროგრამით ხდება **ToLower()** და **ToUpper()** ფუნქციებთან მუშაობის დემონსტრირება.

```
{
String^ striqoni1 = "ANA SABA";
String^ striqoni2 = "lika romani";

label1->Text = striqoni1->ToLower();          //      "ana saba"
label2->Text = striqoni2->ToUpper();          //      "LIKA ROMANI"
}
```

სტრიქონების მასივი

ქვემოთ მოცემული პროგრამით ხდება striqonebis_masivi სტრიქონების მასივის გამოცხადება და ინიციალიზება. ის სამ სტრიქონულ ლიტერალს შეიცავს.

```
{
//      სტრიქონების მასივთან მუშაობის დემონსტრირება
array<String^>^ striqonebis_masivi = { L"სტრიქონული ", L"მასივის ", L"მაგალითი" };
//      მასივის გამოტანა label კომპონენტში
for ( int indexi = 0; indexi < striqonebis_masivi->Length; indexi++ )
        label1->Text += striqonebis_masivi[indexi] + " ";
//      სტრიქონული მასივის მნიშვნელობების შეცვლა
striqonebis_masivi[0] = L"მეორე ";
striqonebis_masivi[1] = L"სტრიქონული ";
striqonebis_masivi[2] = textBox1->Text;
label1->Text += L'\n';
//      მასივის გამოტანა label კომპონენტში
for each ( String^ striqoni in striqonebis_masivi )
        label1->Text += striqoni + " ";
}
```

სტრიქონის უცვლელობა

String ტიპის ობიექტის ერთ-ერთი მნიშვნელოვანი თავისებურებაა ის, რომ სტრიქონში ერთხელ შექმნილი სიმბოლოების მიმდევრობა აღარ შეიცვლება, ე.ი. არ შეიძლება ინდექსის გამოყენება სტრიქონის რომელიმე სიმბოლოსათვის ახალი მნიშვნელობის მისანიჭებლად. მაგალითად, დაუშვებელია ასეთი მინიჭება:

```
striqoni1[1] = L'ს';
```

იმისათვის, რომ სტრიქონის ზოგიერთი სიმბოლო შევცვალოთ ახალი სიმბოლოთი, საჭირო ხდება ახალი სტრიქონის შექმნა, რომელიც საჭირო ცვლილებებს შეიცავს. ამისათვის, შეგვიძლია Replace ან Substring ფუნქციის გამოყენება. ქვემოთ მოცემული პროგრამით ხდება სტრიქონის შეცვლის დემონსტრირება.

```
{
//      სტრიქონის შეცვლის დემონსტრირება
```

```
String^ str1 = L"დაპროგრამება";
String^ str2;
String^ str3;
//        str1[0] = 'ბ';                    // ასეთი მინიჭება დაუშვებელია
str2 = str1->Replace(L'ა', L'ბ');           // str2 სტრიქონში ჩაიწერება შეცვლილი სტრიქონი
str3 = str1->Substring(2, 3);               // str3 სტრიქონში ჩაიწერება ახალი სტრიქონი
str1 = str1->Replace(L"პროგ", L"აბგ");      // ასეთი მინიჭება დასაშვებია. ამ შემთხვევაში
                                            // str1 სტრიქონშივე ჩაიწერება შეცვლილი სტრიქონი
label1->Text = str1;
label2->Text = str2;
label3->Text = str3;
}
```

დინამიკური სტრიქონი

StringBuilder კლასი წარმოგვიდგენს ***დინამიკურ სტრიქონს*** ანუ სიმბოლოების ცვლად სტრიქონს. დინამიკური სტრიქონი შეიძლება შეიცვალოს მასში სიმბოლოების ჩამატების, დამატების, წაშლისა და შეცვლის გზით. დინამიკურ სტრიქონთან სამუშაოდ გამოიყენება System::Text::StringBuilder კლასი. StringBuilder კლასის ფუნქციები მუშაობენ უფრო სწრაფად, რადგან ისინი არ ქმნიან სტრიქონის ასლს და უშუალოდ სტრიქონთან მუშაობენ. დინამიკურ სტრიქონში, String კლასის სტრიქონისაგან განსხვავებით, შესაძლებელია სიმბოლოების პირდაპირ შეცვლა. პროგრამის დასაწყისში, რომელიც დინამიკურ სტრიქონებთან მუშაობს, უნდა მოვათავსოთ დირექტივა:

```
using namespace System::Text;
```

დინამიკური სტრიქონის შექმნა

StringBuilder კლასის კონსტრუქტორი გადატვირთულია, ამიტომ არსებობს ამ კლასის კონსტრუქტორის რამდენიმე ვარიანტი.

ჩვენ განვიხილავთ დინამიკური სტრიქონის შექმნის რამდენიმე გზას. ერთ-ერთი გზაა ცარიელი დინამიკური სტრიქონის შექმნა:

```
StringBuilder^ DinamiuriStriqoni1 = gcnew StringBuilder();
```

თუ კოდის კომპილაციისას გამოჩნდა შეტყობინება იმის შესახებ, რომ StringBuilder კლასი არ არის განსაზღვრული, მაშინ ის უნდა მივუთითოთ შემდეგნაირად:

```
System::Text::StringBuilder^
```

შექმნისას დინამიკური სტრიქონის ტევადობაა 16 სიმბოლო. ახალი სიმბოლოს დამატების შემდეგ დინამიკური სტრიქონის ტევადობა ავტომატურად იზრდება. ტევადობა შეგვიძლია მივუთითოთ დინამიკური სტრიქონის შექმნისას. მაგალითად,

```
{
int Tevadoba = 25;
StringBuilder^ DinamiuriStriqoni2 = gcnew StringBuilder(Tevadoba);
}
```

დინამიკური სტრიქონის შექმნისას კონსტრუქტორს შეგვიძლია გადავცეთ, აგრეთვე, მაქსიმალური ტევადობა. მაგალითად,

```
{
int Tevadoba = 25;
```

```
int MaxTevadoba = 100;
StringBuilder^ DinamiuriStriqoni3 = gcnew StringBuilder(Tevadoba, MaxTevadoba);
}
```

შექმნისას დინამიკურ სტრიქონს ავტომატურად ენიჭება 2147483647-ის ტოლი მაქსიმალური ტევადობა.

დინამიკური სტრიქონის შექმნისას კონსტრუქტორს შეგვიძლია გადავცეთ სტრიქონი, რომელიც დინამიკურ სტრიქონში უნდა ჩაიწეროს. მაგალითად,

```
{
String^ striqoni = L"ანა და საბა";
StringBuilder^ DinamiuriStriqoni4 = gcnew StringBuilder(striqoni);
}
```

დინამიკური სტრიქონის შექმნისას კონსტრუქტორს შეგვიძლია გადავცეთ სტრიქონი; საწყისი ინდექსი, საიდანაც უნდა დაიწყოს სიმბოლოების გადაწერა; გადასაწერი სიმბოლოების რაოდენობა და დინამიკური სტრიქონის ტევადობა:

```
{
String^ Striqoni = L"ანა და საბა";
int Tevadoba = 50;
int SawyisiIndexi = 4;
int SimboloebisRaodenoba = 7;
StringBuilder^ DinamiuriStriqoni5 =
                gcnew StringBuilder(Striqoni, SawyisiIndexi, SimboloebisRaodenoba, Tevadoba);
label1->Text = DinamiuriStriqoni5->ToString();          //        DinamiuriStriqoni5 = "და საბა"
}
```

ცხრილი 11.4. StringBuilder კლასის თვისებები

თვისება	ტიპი	აღწერა
Capacity	int	გასცემს ან აყენებს დინამიკური სტრიქონის ტევადობას
Length	int	გასცემს ან აყენებს დინამიკურ სტრიქონში სიმბოლოების რაოდენობას
MaxCapacity	int	გასცემს დინამიკური სტრიქონის მაქსიმალურ ტევადობას

ცხრილი 11.5. StringBuilder კლასის ფუნქციები

ფუნქცია	დაბრუნებული მნიშვნელობის ტიპი	აღწერა
Append(Object^ *ობიექტი*)	StringBuilder^	ობიექტის სტრიქონულ წარმოდგენას ამატებს დინამიკური სტრიქონის ბოლოში
AppendFormat(String^ *ფორმატი*, Object^ *ობიექტი*)	StringBuilder^	დაფორმატებულ სტრიქონს ამატებს დინამიკური სტრიქონის ბოლოში
EnsureCapacity(int *ტევადობა*)	int	ამოწმებს, ტოლია თუ მეტი დინამიკური სტრიქონის მიმდინარე ტევადობა მითითებულ მნიშვნელობაზე

ცხრილი 11.5. (გაგრძელება)

Equals(Object^ *ობიექტი*)	bool	გასცემს true მნიშვნელობას თუ დინამიკური სტრიქონი მითითებული ობიექტის ტოლია, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას
GetType()	Type^	გასცემს ობიექტის ტიპს
Insert(Object^ *ობიექტი*)	StringBuilder^	მითითებული ობიექტის სტრიქონულ წარმოდგენას ამატებს დინამიკურ სტრიქონში დაწყებული მითითებული პოზიციიდან
Remove(int *საწყისი_ინდექსი*, int *სიგრძე*)	StringBuilder^	დინამიკური სტრიქონიდან შლის მითითებული რაოდენობის სიმბოლოს დაწყებული მითითებული პოზიციიდან
Replace(String^ *ძველი_სტრიქონი*, String^ *ახალი_სტრიქონი*)	StringBuilder^	მითითებულ სიმბოლოს ან ქვესტრიქონს დინამიკურ სტრიქონში ყველგან ცვლის მითითებული სიმბოლოთი ან ქვესტრიქონით.
ToString()	String^	დინამიკურ სტრიქონს გარდაქმნის სტრიქონად

StringBuilder კლასის თვისებები და ფუნქციები

ამ კლასის თვისებები და ფუნქციები მოცემულია 11.4 და 11.5 ცხრილებში.

განვიხილოთ ზოგიერთი მათგანი.

Append() და **AppendFormat()** ფუნქციები. Append() ფუნქცია მითითებულ ობიექტს სტრიქონად გარდაქმნის და ამატებს მას დინამიკურ სტრიქონისს. ობიექტს შეიძლება ჰქონდეს ნებისმიერი ჩადგმული ტიპი, როგორიცაა, bool, byte, int, double და ა.შ. Append() ფუნქციის უმარტივეს ვერსიას აქვს შემდეგი სინტაქსი:

StringBuilder^ Append(*მნიშვნელობა*)

აქ *მნიშვნელობას* შეიძლება ჰქონდეს ნებისმიერი ტიპი.

მოცემული პროგრამით ხდება სხვადასხვა ტიპის ობიექტების დინამიკური სტრიქონისათვის დამატების დემონსტრირება.

```
{
//      დინამიკური სტრიქონისათვის ელემენტების დამატება
String^ striqoni = L"საბა და ანა";
int    ricxvi = 5;
double wiladi = 2.8765;
bool logikuri = true;
StringBuilder^ DinamiuriStriqoni = gcnew StringBuilder();

DinamiuriStriqoni->Append(striqoni);        //      DinamiuriStriqoni = "საბა და ანა"
DinamiuriStriqoni->Append(wiladi);          //      DinamiuriStriqoni = "საბა და ანა2.8765"
DinamiuriStriqoni->Append(logikuri);        //      DinamiuriStriqoni = "საბა და ანა2.8765true"
DinamiuriStriqoni->Append(ricxvi);          //      DinamiuriStriqoni = "საბა და ანა2.8765true5"
label1->Text = DinamiuriStriqoni->ToString();
}
```

Append() ფუნქციის მეორე ვერსია საშუალებას გვაძლევს დინამიკურ სტრიქონს დავუმატოთ გამეორებადი სიმბოლოს სერია. ფუნქციის სინტაქსია:

StringBuilder^ Append(Char *სიმბოლო*, int *გამეორების_რაოდენობა*)

მაგალითად, DinamiuriStriqoni->Append(L'ა', 10); ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკურ სტრიქონს დაემატება 10 ცალი 'ა' სიმბოლო.

დინამიკურ სტრიქონს შეგვიძლია დავუმატოთ მითითებული სტრიქონის მითითებული სიმბოლოები. შესაბამისი ფუნქციის სინტაქსია:

StringBuilder^ Append(
String^ ***სტრიქონი,*** **int** ***საწყისი_ინდექსი,*** **int** ***სიმბოლოების_რაოდენობა*****)**

მაგალითად, DinamiuriStriqoni->Append(L"ანა და საბა", 4, 7); ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკურ სტრიქონს დაემატება სტრიქონი "და საბა".

AppendFormat() ფუნქცია ამატებს დინამიკური სტრიქონისათვის დაფორმატებულ სტრიქონს. მისი სინტაქსია:

StringBuilder^ AppendFormat(String^ ***ფორმატი,*** **Object^** ***ობიექტი*****)**

აქ *ობიექტი* წარმოდგენილი იქნება *ფორმატის* მიხედვით და შემდეგ დაემატება დინამიკურ სტრიქონს. მაგალითად:

```
{
StringBuilder^ DinamiuriStriqoni = gcnew StringBuilder();
double wiladi = 2.8765;
DinamiuriStriqoni->AppendFormat("{0, 5:f2}", wiladi);
label1->Text = DinamiuriStriqoni->ToString();
}
```

AppendFormat() ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკურ სტრიქონს დაემატება სტრიქონი " 2.88". ფორმატში რიცხვი 5 არის ათწილადში თანრიგების რაოდენობა მძიმის ჩათვლით, რიცხვი 2 კი - თანრიგების რაოდენობა წილად ნაწილში. შედეგად, ათწილადის მთელ ნაწილს დაეთმობა 2 თანრიგი, მძიმეს ერთი, წილად ნაწილს კი - 2. 0 მიუთითებს, რომ უნდა დაფორმატდეს პირველი ცვლადი, კერძოდ კი - wiladi.

Insert() ფუნქცია ობიექტს სტრიქონად გარდაქმნის და დინამიკურ სტრიქონში ამატებს მითითებული პოზიციიდან. Insert() ფუნქციის უმარტივეს ვერსიას აქვს შემდეგი სინტაქსი:

StringBuilder^ Insert(**int** ***ინდექსი,*** **String^** ***სტრიქონი*****)**

მაგალითად, DinamiuriStriqoni->Insert(5, L"საბა, "); ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკურ სტრიქონს მე-5 პოზიციიდან ჩაემატება "საბა, " სტრიქონი.

დინამიკურ სტრიქონში შეგვიძლია ჩავსვათ, აგრეთვე, ჩასასმელი სტრიქონის რამდენიმე ასლი. Insert() ფუნქციის ამ ვერსიის სინტაქსია:

StringBuilder^ Insert(**int** ***ინდექსი,*** **String^** ***სტრიქონი,*** **int** ***გამეორების_რაოდენობა*****)**

მაგალითად, DinamiuriStriqoni->Insert(5, "საბა, ", 4); ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკურ სტრიქონს მე-5 პოზიციიდან 4-ჯერ ჩაემატება "საბა, " სტრიქონი.

Remove() ფუნქცია. ის დინამიკური სტრიქონის მითითებული პოზიციიდან შლის მითითებული რაოდენობის სიმბოლოს. ფუნქციის სინტაქსია:

StringBuilder^ Remove(int ***საწყისი_ინდექსი,*** **int** ***სიმბოლოების_რაოდენობა*****)**

მაგალითი.

```
{
StringBuilder^ DinamiuriStriqoni1 = gcnew StringBuilder(L"ანა და საბა");
DinamiuriStriqoni1->Remove(3, 4);                    // DinamiuriStriqoni1 = "ანასაბა"
label1->Text = L"DinamiuriStriqoni1 = " + DinamiuriStriqoni1->ToString();
}
```

Replace() ფუნქცია. ის მითითებულ სიმბოლოს ან სტრიქონს დინამიკურ სტრიქონში ყველგან ცვლის ახალი სიმბოლოთი ან სტრიქონით. ფუნქციის სინტაქსია:

StringBuilder^ Replace(String^ ***ძველი_მნიშვნელობა,*** **String^** ***ახალი_მნიშვნელობა*****)**

```
{
```

```
StringBuilder^ DinamiuriStriqoni1 = gcnew StringBuilder(L"ანა და საბა");
DinamiuriStriqoni1->Replace(L'ა', L'რ');          //  DinamiuriStriqoni1 = "რნრ დრ სრბრ"
label1->Text += L"DinamiuriStriqoni1 = " + DinamiuriStriqoni1->ToString();
}
```

ToString() ფუნქცია. ის დინამიკურ სტრიქონს გარდაქმნის ჩვეულებრივ სტრიქონად. ფუნქციის სინტაქსია:

virtual String^ ToString() override

მაგალითად, String^ striqoni = DinamiuriStriqoni->ToString(); ფუნქციის შესრულების შედეგად DinamiuriStriqoni დინამიკური სტრიქონი გარდაიქმნება ჩვეულებრივ სტრიქონად.

II ნაწილი. C++ ენის სხვა შესაძლებლობები

თავი 13. თარიღი, დრო და დროის ინტერვალი

თარიღი და დრო

თარიღთან და დროსთან სამუშაოდ გამოიყენება System::DateTime სტრუქტურა. მისი თვისებები და ფუნქციები მოცემულია 12.1 და 12.2 ცხრილებში. რაც შეეხება Ticks თვისებას, მის ქართულ შესატყვისად შევარჩიე სიტყვა "წიკი", საათის წიკწიკიდან გამომდინარე (რუსულად тик). წიკი არის 100-ნანოწამიანი ინტერვალი.

არსებობს DateTime სტრუქტურის კონსტრუქტორის რამდენიმე ვერსია. შესაბამისად, DateTime სტრუქტურის ობიექტი (ეგზემპლარი) შეგვიძლია რამდენიმე საშუალებით შევქმნათ. ერთ-ერთია კონსტრუქტორისთვის წლის, თვის და დღის გადაცემა:

```
{
int weli = 2005;
int tve  = 3;
int dge  = 19;
//        იქმნება Tarigi სტრუქტურა, რომელიც შეიცავს წელს, თვესა და დღეს
DateTime^ Tarigi = gcnew DateTime(weli, tve, dge);
label1->Text = Tarigi->Year.ToString() + " " + Tarigi->Month.ToString() + " " + Tarigi->Day.ToString();
}
```

კონსტრუქტორს შეიძლება, აგრეთვე, გადავცეთ საათი, წუთი, წამი და მილიწამი:

```
{
int weli = 2005;
int tve  = 3;
int dge  = 19;
int saati = 20;
int wuti  = 30;
int wami  = 25;
int miliwami = 45;
//        იქმნება Tarigi სტრუქტურა, რომელიც შეიცავს
//        წელს, თვეს, დღეს, საათს, წუთს, წამს და მილიწამს
DateTime^ Tarigi = gcnew DateTime(weli, tve, dge, saati, wuti, wami, miliwami);
label1->Text = Tarigi->Year.ToString() + " " + Tarigi->Month.ToString()  + " " + Tarigi->Day.ToString() +
                  "  " + Tarigi->Hour.ToString()   + " " + Tarigi->Minute.ToString() + "  " +
                   Tarigi->Second.ToString() + " " + Tarigi->Millisecond.ToString();
}
```

DateTime სტრუქტურის კონსტრუქტორს უკანასკნელ პარამეტრად შეიძლება გადავცეთ კალენდარი - System::Globalization::Calendar კლასის ობიექტი. Calendar კლასი უზრუნველყოფს დროის დაყოფას ინტერვალებად, კერძოდ წლებად, თვეებად და კვირეებად. არსებობს რამდენიმე კლასი, რომლებიც ახდენენ Calendar კლასის ფუნქციების რეალიზებას. ეს კლასებია: EastAsianLunisolarCalendar, GregorianCalendar, HebrewCalendar, HijriCalendar, JapaneseCalendar, JapaneseLunisolarCalendar, JulianCalendar, KoreanCalendar, KoreanLunisolarCalendar, Persian-Calendar, TaiwanCalendar, TaiwanLunisolarCalendar, ThaiBuddhistCalendar, UmAlQuraCalendar.

მოცემულ მაგალითში კონსტრუქტორს გადაეცემა JulianCalendar კლასის ობიექტი:

```
{
System::Globalization::JulianCalendar^ Kalendari = gcnew System::Globalization::JulianCalendar();
System::DateTime^ Tarigi = gcnew System::DateTime(2005, 3, 19, Kalendari);
label1->Text = Tarigi->Year.ToString() + "  " + Tarigi->Month.ToString() + "  " + Tarigi->Day.ToString();
}
```

ცხრილი 12.1. DateTime სტრუქტურის თვისებები

თვისება	ტიპი	აღწერა
Now (სტატიკურია)	DateTime	შეიცავს მიმდინარე თარიღსა და დროს
Today (სტატიკურია)	DateTime	შეიცავს მიმდინარე თარიღს. დროის ველები განულებულია 00:00:00
UtcNow() (სტატიკურია)	DateTime	შეიცავს მიმდინარე თარიღსა და დროს დროის სარტყლის გათვალისწინებით
Date	DateTime	შეიცავს თარიღს. დროის ველები განულებულია 00:00:00
Day	int	შეიცავს თვის დღის მნიშვნელობას 1÷31 დიაპაზონში
DayOfWeek	DayOfWeek	შეიცავს კვირის დღის მნიშვნელობას 0 (კვირა)÷ 6 (შაბათი) დიაპაზონში
DayOfYear	int	შეიცავს წლის დღის ნომერს 1÷366 დიაპაზონში
Hour	int	შეიცავს საათის მნიშვნელობას 0÷23 დიაპაზონში
Millisecond	int	შეიცავს მილიწამის მნიშვნელობას 0÷999 დიაპაზონში
Minute	int	შეცავს წუთის მნიშვნელობას 0÷59 დიაპაზონში
Month	int	შეიცავს თვის მნიშვნელობას 1÷12 დიაპაზონში
Second	int	შეიცავს წამის მნიშვნელობას 0÷59 დიაპაზონში
Ticks	long long	შეიცავს წიკების (100-ნანოწამიანი ინტერვალების) რაოდენობას, რომელიც გავიდა 0001 წლის 1 იანვრიდან ეგზემპლარში დაყენებულ დრომდე
TimeOfDay	TimeSpan	შეიცავს შუაღამიდან გასულ დროს
Year	int	შეიცავს წლის მნიშვნელობას 1÷9999 დიაპაზონში

ახლა განვიხილოთ ზოგიერთი თვისება და ფუნქცია.

Now და **UtcNow** თვისებები. Now თვისება შეიცავს კომპიუტერის სისტემური საათიდან აღებულ თარიღსა და დროს. UtcNow თვისება შეიცავს თარიღსა და დროს UTC (Universal Coordinated Time) ფორმატში, რომელსაც, აგრეთვე, გრინვიჩის დროს (GMT - Greenwich Mean Time) უწოდებენ. ის შეესაბამება ინგლისის ერთ-ერთი დაბის - გრინვიჩის დროის სარტყელს. სტანდარტული წყნარი ოკეანის დრო (PST - Pacific Standard Time) გრინვიჩის დროს რვა საათით ჩამორჩება. მოცემული პროგრამით ხდება ამ თვისებებთან მუშაობის დემონსტრირება.

```
{
//      Now და UtcNow თვისებებთან მუშაობის დემონსტრირება
DateTime^ Tarigi_Dro1 = DateTime::Now;
DateTime^ Tarigi_Dro2 = DateTime::UtcNow;
label1->Text = Tarigi_Dro1->Day.ToString() + " " + Tarigi_Dro1->Month.ToString() + " "
                + Tarigi_Dro1->Year.ToString() + "\n";
label1->Text += Tarigi_Dro2->Day.ToString() + " " + Tarigi_Dro2->Month.ToString() + " "
                + Tarigi_Dro2->Year.ToString();
}
```

ცხრილი 12.2. DateTime სტრუქტურის ფუნქციები

ფუნქცია	დასაბრუნებელი ტიპი	აღწერა
Compare(DateTime *თარიღი_დრო1*, DateTime *თარიღი_დრო2*) (სტატიკურია)	int	ადარებს DateTime სტრუქტურის ორ ეგზემპლარს.
DaysInMonth(int *წელი*, int *თვე*) (სტატიკურია)	int	გასცემს დღეების რაოდენობას მითითებული წლის მითითებული თვისათვის
Equals(DateTime *თარიღი_დრო1*, DateTime *თარიღი_დრო2*)	bool	ტოლობაზე ადარებს ორ DateTime სტრუქტურას
FromFileTime(long long *ფაილის_დრო*) (სტატიკურია)	DateTime	გასცემს DateTime სტრუქტურის ეგზემპლარს, რომელიც ფაილის დროითი ჭდის ეკვივალენტურია
FromOADate(double პარამეტრი) (სტატიკურია)	DateTime	გასცემს DateTime სტრუქტურის ეგზემპლარს, რომელიც OLE თარიღის ეკვივალენტურია
IsLeapYear(int *წელი*) (სტატიკურია)	bool	გასცემს true-ს, თუ წელი ნაკიანია, წინააღმდეგ შემთხვევაში false-ს
Parse(String^ *თარიღი_დრო*) (სტატიკურია)	DateTime	თარიღისა და დროის სტრიქონულ წარმოდგენას გარდაქმნის DateTime სტრუქტურის ეკვივალენტურ ეგზემპლარად
Add(TimeSpan *დროის_ინტერვალი*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს TimeSpan კლასის ეგზემპლარს
AddDays(double *დღეების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს დღეების მითითებულ რაოდენობას
AddHours(double *საათების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს საათების მითითებულ რაოდენობას
AddMilliseconds(double *მილიწამების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს მილიწამების მითითებულ რაოდენობას
AddMinutes(double *წუთების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს წუთების მითითებულ რაოდენობას
AddMonths(int *თვეების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს თვეების მითითებულ რაოდენობას
AddSeconds(double *წამების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს წამების მითითებულ რაოდენობას
AddTicks(long long *წიკების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს წიკების მითითებულ რაოდენობას
AddYears(int *წლების_რაოდენობა*)	DateTime	DateTime სტრუქტურის ეგზემპლარს უმატებს წლების მითითებულ რაოდენობას

ცხრილი 12.2. (გაგრძელება)

CompareTo(DateTime *პარამეტრი*)	int	DateTime სტრუქტურის ეგზემპლარს ადარებს მითითებულ ობიექტთან.
GetDateTimeFormats()	array<String^>^	DateTime სტრუქტურის ეგზემპლარს გარდაქმნის სტრიქონების მასივად, რომელიც შეიცავს ამ სტრუქტურის ყველა შესაძლო ფორმატს
GetType()	Type^	გასცემს მიმდინარე ობიექტის ტიპს
Subtract(DateTime *პარამეტრი*)	TimeSpan	DateTime სტრუქტურის ეგზემპლარს აკლებს ამავე სტრუქტურის ეგზემპლარს ან TimeSpan კლასის ეგზემპლარს
ToFileTime()	long long	DateTime სტრუქტურის ეგზემპლარს გარდაქმნის ფაილის დროით ჭდედ
ToLocalTime()	DateTime	გრინვიჩის დროს გარდაქმნის ადგილობრივ დროდ
ToLongDateString()	String^	DateTime სტრუქტურის ეგზემპლარის თარიღს გარდაქმნის გრძელი თარიღის შემცველ სტრიქო-ნად
ToLongTimeString()	String^	DateTime სტრუქტურის ეგზემპლარის დროს გარდაქმნის გრძელი დროის შემცველ სტრიქონად
ToOADate()	double	DateTime სტრუქტურის ეგზემპლარს გარდაქმნის ეკვივალენტურ OLE თარიღად
ToShortDateString()	String^	DateTime სტრუქტურის ეგზემპლარის თარიღს გარდაქმნის მოკლე თარიღის შემცველ სტრიქონად
ToShortTimeString()	String^	DateTime სტრუქტურის ეგზემპლარის დროს გარდაქმნის მოკლე დროის შემცველ სტრიქონად
ToString()	String^	DateTime სტრუქტურის ეგზემპლარს გარდაქმნის ეკვივალენტურ სტრიქონად
ToUniversalTime()	DateTime	ადგილობრივ დროს გარდაქმნის გრინვიჩის დროდ

მოცემული პროგრამით ხდება **Date, Day, DayOfWeek, DayOfYear** და **TimeOfDay** თვისებებთან მუშაობის დემონსტრირება.

```
{
//      Date, Day, DayOfWeek, DayOfYear, TimeOfDay
//      თვისებებთან მუშაობის დემონსტრირება
DateTime^ Tarigi_Dro1 = DateTime::Now;
label1->Text = Tarigi_Dro1->Date.ToString() + "\n";
label1->Text += Tarigi_Dro1->Day.ToString() + "\n";
label1->Text += Tarigi_Dro1->DayOfWeek.ToString() + "\n";
label1->Text += Tarigi_Dro1->DayOfYear.ToString() + "\n";
label1->Text += Tarigi_Dro1->TimeOfDay.ToString() + "\n";
}
```

Compare() ფუნქცია ადარებს DateTime სტრუქტურის ორ ეგზემპლარს. თუ პირველი ეგზამპლარი მეტია მეორეზე, მაშინ გაიცემა 1. თუ პირველი ეგზემპლარი ნაკლებია მეორეზე, მაშინ გაიცემა -1. თუ ეგზემპლარები ტოლია, მაშინ გაიცემა 0. ფუნქციის სინტაქსია:

DateTime::Compare(*თარიღი_დრო1, თარიღი_დრო2*)

აქ *თარიღი_დრო1* და *თარიღი_დრო2* არის DateTime სტრუქტურის ეგზემპლარები, რომელთა შედარებაც ხდება. მაგალითი:

```
{
DateTime Tarigi_Dro1( DateTime::Now );
DateTime Tarigi_Dro2( 2007, 1, 27 );
int Shedegi = DateTime::Compare(Tarigi_Dro1, Tarigi_Dro2);
switch ( Shedegi )
{
case -1 : label1->Text = L"პირველი თარიღი ნაკლებია მეორეზე"; break;
case  0 : label1->Text = L"თარიღები ტოლია"; break;
case  1 : label1->Text = L" პირველი თარიღი მეტია მეორეზე "; break;
}
}
```

DateTime სტრუქტურის ეგზემპლარების შედარება შეიძლება, აგრეთვე, შედარების გადატვირთული ოპერატორებით: >, >=, <, <=, ==, != . მაგალითად,

```
if ( Tarigi_Dro1 == Tarigi_Dro2 ) label1->Text = L"თარიღები ტოლია";
        else label1.Text = "თარიღები არ არის ტოლი";
```

Equals() ფუნქცია ადარებს DateTime სტრუქტურის ორ ეგზემპლარს. თუ ისინი ტოლია, მაშინ გაიცემა true, წინააღმდეგ შემთხვევაში - false. ამ ფუნქციას აქვს როგორც სტატიკური, ისე ჩვეულებრივი ვერსია. სტატიკური ვერსიის სინტაქსია:

DateTime::Equals(*თარიღი_დრო1, თარიღი_დრო2*)

აქ *თარიღი_დრო1* და *თარიღი_დრო2* არის DateTime სტრუქტურის ეგზემპლარები. მაგალითი.

```
{
DateTime Tarigi_Dro1(DateTime::Now);
DateTime Tarigi_Dro2(2007, 1, 27);
bool Shedegi = DateTime::Equals(Tarigi_Dro1, Tarigi_Dro2);
if ( Shedegi == true ) label1->Text = L"თარიღები ტოლია";
  else label1->Text = L"თარიღები არ არის ტოლი";
}
```

Equals() ფუნქციის ჩვეულებრივი ვერსიის სინტაქსია:

***dateTime*.Equals(*თარიღი_დრო2*)**

აქ *dateTime* არის DateTime სტრუქტურის ობიექტი. მაგალითი:

```
{
DateTime Tarigi_Dro1 = DateTime::Now;
DateTime Tarigi_Dro2(2007, 1, 27);
bool Shedegi = Tarigi_Dro1.Equals(Tarigi_Dro2);
if ( Shedegi == true ) label1->Text = L"თარიღები ტოლია";
        else label1->Text = L"თარიღები არ არის ტოლი";
}
```

DaysInMonth() ფუნქცია გასცემს მითითებულ თვეში დღეების რაოდენობას. მისი სინტაქსია:

DateTime::DaysInMonth(*წელი, თვე*)

მაგალითი:

```
{
int DgeebisRaodenoba = DateTime::DaysInMonth(2007, 3);
label1->Text = DgeebisRaodenoba.ToString();                    // DgeebisRaodenoba = 31
}
```

IsLeapYear() ფუნქცია გასცემს true-ს თუ თვე ნაკიანია და false-ს წინააღმდეგ შემთხვევაში. მისი სინტაქსია:

DateTime::IsLeapYear(*წელი*)

მაგალითი:

```
{
bool ArisNakiani = DateTime::IsLeapYear(2009);
if ( ArisNakiani == true ) label1->Text = L"წელი ნაკიანია";
   else label1->Text = L" წელი არ არის ნაკიანი";
}
```

Parse() ფუნქცია თარიღისა და დროის შემცველ სტრიქონს გარდაქმნის DateTime სტრუქტურის ეკვივალენტურ ეგზემპლარად. მისი სინტაქსია:

DateTime::Parse(*სტრიქონი*)

მაგალითი:

```
{
DateTime Tarigi_Dro1 = DateTime::Parse("10/5/2000");
DateTime Tarigi_Dro2 = DateTime::Parse("10/5/2000 8:30:35");
label1->Text = Tarigi_Dro1.ToString() + "   " + Tarigi_Dro2.ToString();
}
```

Add() და **Subtract()** ფუნქციები. Add() ფუნქცია DateTime სტრუქტურის ეგზემპლარს უმატებს TimeSpan კლასის ობიექტს. Subtract() ფუნქცია DateTime სტრუქტურის ეგზემპლარს აკლებს TimeSpan კლასის ობიექტს. ამ ფუნქციების სინტაქსია:

***dateTime*.Add(*დროის_ინტერვალი*)**

***dateTime*.Subtract(*დროის_ინტერვალი*)**

აქ *დროის_ინტერვალი* არის TimeSpan სტრუქტურის ობიექტი. მაგალითი:

```
{
TimeSpan Drois_Intervali(1, 5, 10, 20);
DateTime Tarigi_Dro1(2007, 5, 15, 17, 35, 30);
DateTime Tarigi_Dro2 = Tarigi_Dro1.Add(Drois_Intervali);                  // 16.05.2007 22:45:50
DateTime Tarigi_Dro3 = Tarigi_Dro1.Subtract(Drois_Intervali);             // 14.05.2007 12:25:10
label1->Text = Tarigi_Dro2.ToString() + "  " + Tarigi_Dro3.ToString();
}
```

თავდაპირველად პროგრამით იქმნება დროის ინტერვალი: 1 დღე, 5 საათი, 10 წუთი და 20 წამი. შემდეგ იქმნება Tarigi_Dro1 სტრუქტურა: 2007 წლის 15 მაისი, 17 საათი, 35 წუთი და 30 წამი. Add() ფუნქცია Tarigi_Dro1 სტრუქტურას დაუმატებს Drois_Intervali დროის ინტერვალს. შედეგად, Tarigi_Dro2 სტრუქტურას მიენიჭება მნიშვნელობა - 16.05.2007 22:45:50. Subtract() ფუნქცია Tarigi_Dro1 სტრუქტურას გამოაკლებს Drois_Intervali დროის ინტერვალს. შედეგად, Tarigi_Dro3 სტრუქტურას მიენიჭება მნიშვნელობა - 14.05.2007 12:25:10.

იგივე ოპერაციები შეგვიძლია შევასრულოთ გადატვირთული "+" და "-" ოპერატორების გამოყენებით. მაგალითი:

```
DateTime Tarigi_Dro4 = Tarigi_Dro1 + Drois_Intervali;
DateTime Tarigi_Dro5 = Tarigi_Dro1 - Drois_Intervali;
```

ცხრილი 12.3. საფორმატო სტრიქონის კომპონენტები

ფორმატის ელემენტი	აღწერა
d	თვის დღე წინა ნულის გარეშე დღის ნომრებისათვის, რომლებიც ერთი ციფრისგან შედგება
Dd	თვის დღე წინა ნულით დღის ნომრებისათვის, რომლებიც ერთი ციფრისგან შედგება
ddd	კვირის დღის შემოკლებული დასახელება
dddd	კვირის დღის სრული დასახელება
f	წამის ნაწილი ერთი ციფრის სიზუსტით. სიზუსტის გასაზრდელად შეგვიძლია გამოვიყენოთ შვიდი "f" სიმბოლო. დანარჩენი ციფრები არ გამოჩნდება.
M	თვის ნომერი წინა ნულის გარეშე თვის ნომრებისათვის, რომლებიც ერთი ციფრისაგან შედგება
MM	თვის ნომერი წინა ნულით თვის ნომრებისათვის, რომლებიც ერთი ციფრისაგან შედგება
MMM	თვის შემოკლებული დასახელება
MMMM	თვის სრული დასახელება
y	წელი ასწლეულის გარეშე. 10-ზე ნაკლები მნიშვნელობა აისახება წინა ნულის გარეშე
Yy	წელი ასწლეულის გარეშე. 10-ზე ნაკლები მნიშვნელობა აისახება წინა ნულით
yyyy	წლის სრული ნომერი შემდგარი ოთხი ციფრისგან
Gg	ერა. ეს ელემენტი იგნორირდება თუ თარიღი არ შეიცავს ერას
h	საათი თორმეტსაათიანი წარმოდგენით, წინა ნულის გარეშე, ერთი ციფრისგან შემდგარი საათებისთვის
Hh	საათი თორმეტსაათიანი წარმოდგენით, წინა ნულით, ერთი ციფრისგან შემდგარი საათებისთვის
H	საათი, ოცდაოთხსაათიანი წარმოდგენით, წინა ნულის გარეშე, ერთი ციფრისგან შემდგარი საათებისთვის
HH	საათი, ოცდაოთხსაათიანი წარმოდგენით, წინა ნულით, ერთი ციფრისგან შემდგარი საათებისთვის
m	წუთები, წინა ნულის გარეშე, ერთი ციფრისგან შემდგარი წუთებისთვის
mm	წუთები, წინა ნულით, ერთი ციფრისგან შემდგარი წუთებისთვის
s	წამები, წინა ნულის გარეშე, ერთი ციფრისგან შემდგარი წამებისთვის
ss	წამები, წინა ნულით, ერთი ციფრისგან შემდგარი წამებისთვის
t	A.M./P.M მიმთითებლის პირველი სიმბოლო
tt	A.M./P.M სრული მიმთითებელი
zz	დროის სარტყლის წანაცვლება, წინა ნულით, ერთი ციფრისგან შემდგარი წანაცვლებისთვის
zzz	სასაათო სარტყლის წანაცვლება შემდგარი საათებისა და წუთებისაგან. ერთი ციფრისგან შემდგარი მნიშვნელობები შეივსება ნულით

AddYears(), AddMonths(), AddDays(), AddHours() და **AddMinutes()** ფუნქციები. ეს ფუნქციები DateTime სტრუქტურის ეგზემპლარს შესაბამისად უმატებს წლებს, თვეებს, დღეებს, საათებსა და წუთებს. ამ ფუნქციებს აქვთ double ტიპის პარამეტრი. მაგალითი:

```
{
DateTime Tarigi_Dro1 = DateTime::Now;
```

```
label1->Text = Tarigi_Dro1.ToString() + "\n";
Tarigi_Dro1 = Tarigi_Dro1.AddYears(3);
Tarigi_Dro1 = Tarigi_Dro1.AddMonths(2);
Tarigi_Dro1 = Tarigi_Dro1.AddDays(10);
Tarigi_Dro1 = Tarigi_Dro1.AddHours(7);
Tarigi_Dro1 = Tarigi_Dro1.AddMinutes(40);
label1->Text += Tarigi_Dro1.ToString();
}
```

მოცემული პროგრამით ხდება **ToLongDateString(), ToShortDateString(), ToLongTimeString()** და **ToShortTimeString()** ფუნქციების გამოყენების დემონსტრირება.

```
{
DateTime Tarigi_Dro1 = DateTime::Now;
label1->Text  = Tarigi_Dro1.ToLongDateString() + "\n";
label1->Text += Tarigi_Dro1.ToShortDateString()+"\n";
label1->Text += Tarigi_Dro1.ToLongTimeString() + "\n";
label1->Text += Tarigi_Dro1.ToShortTimeString();
}
```

String() ფუნქცია. მისი გადატვირთული ვერსია ახდენს DateTime სტრუქტურის ეგზემპლარის გარდაქმნას ეკვივალენტურ სტრიქონად. მისი სინტაქსია:

***dateTime.*ToString()**

მაგალითი:

```
{
DateTime Tarigi_Dro1 = DateTime::Now;
label1->Text = Tarigi_Dro1.ToString();
}
```

ToString() ფუნქციას შეგვიძლია პარამეტრად გადავცეთ ფორმატი (სტრიქონი), რომლის მიხედვითაც შესრულდება DateTime სტრუქტურის ეგზემპლარის გარდაქმნა. ToString() ფუნქციის ამ ვერსიის სინტაქსია:

***dateTime.*ToString(*ფორმატი*)**

მაგალითი:

```
{
DateTime Tarigi_Dro1 = DateTime::Now;
label1->Text  = Tarigi_Dro1.ToString() + "\n";
label1->Text += Tarigi_Dro1.ToString("MM/dd/yy");
label1->Text += Tarigi_Dro1.ToString("D");
}
```

პროგრამის შესრულების შედეგად მიიღება სტრიქონი, მაგალითად, "02.03.07". აქ 02 არის თვე, 03 - დღე, 07 კი - წელი. კოდის უკანასკნელ სტრიქონში გამოყენებულია ჩადგმული ფორმატებიდან ერთ-ერთი, კერძოდ კი - "D".

12.3 და 12.4 ცხრილებში შესაბამისად მოცემულია საფორმატო სტრიქონის კომპონენტები და თარიღისა და დროის ჩადგმული ფორმატები.

ცხრილი 12.4. თარიღისა და დროის ჩადგმული ფორმატები

სიმბოლო	ფორმატი	მაგალითი
d	MM/dd/yyyy	03.19.2007
D	dddd, MMMM dd, yyyy	2007 წლის 04 02, კვირა
f	dddd, MMMM dd, yyyy HH:mm	2007 წლის 04 02, კვირა 9:36
F	dddd, MMMM dd, yyyy HH:mm:ss	2007 წლის 04 02, კვირა 9:36:51
g	MM/dd/yyyy/ HH:mm	04.02.2007 9:36:51
G	MM/dd/yyyy HH:mm:ss	04.02.2007 9:36
m, M	MMMM dd	04 02
r, R	ddd, dd MM yyyy HH':'mm':'ss 'GMT'	Sun, 04 Feb 2007 09:42:17 GMT
s	yyyy'-'MM'-'dd'T'HH':'mm':'ss	2007-02-04T09:42:17
t	HH:mm	9:42
T	HH:mm:ss	9:42:17
u	yyyy'-'MM'-'dd HH':'mm':'ss'Z'	2007-02-04 09:42:17Z
U	dddd, MMMM dd, yyyy HH:mm:ss	2007 წლის 04 02, კვირა 5:42:17
y, Y	yyyy, MMMM	თებერვალი 2007

დროის ინტერვალი

თარიღისა და დროის გარდა შეგვიძლია, აგრეთვე, ვიმუშაოთ დროის ინტერვალთან. დროის ინტერვალთან სამუშაოდ გამოიყენება System::TimeSpan სტრუქტურა. ამ კლასის კონსტრუქტორს აქვს რამდენიმე გადატვირთული ვერსია. მათი სინტაქსია:

TimeSpan(int *საათი,* int *წუთი,* int *წამი*)
TimeSpan(int *დღე,* int *საათი,* int *წუთი,* int *წამი*)
TimeSpan(int *დღე,* int *საათი,* int წუთი, int *წამი,* int *მილიწამი*)
TimeSpan(int *წიკი*)

მაგალითი.

```
{
//      დროის ინტერვალის შექმნა საათის, წუთისა და წამის მითითებით
TimeSpan Drois_Intervali1(5, 15, 45);
//      დროის ინტერვალის შექმნა დღის, საათის, წუთისა და წამის მითითებით
TimeSpan Drois_Intervali2(2, 5, 15, 45);
//      დროის ინტერვალის შექმნა დღის, საათის, წუთის, წამისა და მილიწამის მითითებით
TimeSpan Drois_Intervali3(2, 5, 15, 45, 80);
//      დროის ინტერვალის შექმნა წიკის მითითებით
TimeSpan Drois_Intervali4(1000);
}
```

TimeSpan კლასის ობიექტი შეგვიძლია დავუმატოთ DateTime სტრუქტურის ეგზემპლარს. მაგალითად,

```
{
DateTime Tarigi_Dro(2009, 04, 12);
TimeSpan Drois_Intervali1(2, 5, 15, 45);
Tarigi_Dro += Drois_Intervali1;                 //      Tarigi_Dro = 14.04.2009 5:15:45
label1->Text = Tarigi_Dro.ToString();
}
```

განვიხილოთ ზოგიერთი თვისება და ფუნქცია.

Days, Hours, Minutes, Seconds, Milliseconds და **Ticks** თვისებებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
// Days, Hours, Minutes, Seconds, Milliseconds და
// Ticks თვისებებთან მუშაობის დემონსტრირება
TimeSpan Drois_Intervali1(5, 15, 45);
label1->Text  = Drois_Intervali1.Days.ToString()  + "\n";
label1->Text += Drois_Intervali1.Hours.ToString() + "\n";
label1->Text += Drois_Intervali1.Minutes.ToString() + "\n";
label1->Text += Drois_Intervali1.Seconds.ToString() + "\n";
label1->Text += Drois_Intervali1.Milliseconds.ToString() + "\n";
label1->Text += Drois_Intervali1.Ticks.ToString() + "\n";
}
```

შედეგში Days და Milliseconds თვისებებს ექნება ნულოვანი მნიშვნელობები, რადგან Drois_Intervali1 ობიექტის შექმნისას მათი მნიშვნელობები არ იყო მითითებული.

TimeSpan კლასის თვისებები და ფუნქციები მოცემულია 12.5 და 12.6 ცხრილებში.

ცხრილი 12.5. TimeSpan კლასის თვისებები

თვისება	ტიპი	აღწერა
Days	int	შეიცავს დღეების რაოდენობას TimeSpan კლასის ობიექტში
Hours	int	შეიცავს საათების რაოდენობას TimeSpan კლასის ობიექტში
Milliseconds	int	შეიცავს მილიწამების რაოდენობას TimeSpan კლასის ობიექტში
Minutes	int	შეიცავს წუთების რაოდენობას TimeSpan კლასის ობიექტში
Seconds	int	შეიცავს წამების რაოდენობას TimeSpan კლასის ობიექტში
Ticks	long long	შეიცავს წიკების რაოდენობას TimeSpan კლასის ობიექტში
TotalDays	double	შეიცავს TimeSpan კლასის ობიექტის ხანგრძლივობას დღეებში
TotalHours	double	შეიცავს TimeSpan კლასის ობიექტის ხანგრძლივობას საათებში
TotalMilliseconds	double	შეიცავს TimeSpan კლასის ობიექტის ხანგრძლივობას მილიწამებში
TotalMinutes	double	შეიცავს TimeSpan კლასის ობიექტის ხანგრძლივობას წუთებში
TotalSeconds	double	შეიცავს TimeSpan კლასის ობიექტის ხანგრძლივობას წამებში

ცხრილი 12.6. TimeSpan კლასის ფუნქციები

ფუნქცია	დაბრუნებული მნიშვნელობის ტიპი	აღწერა
Compare(TimeSpan *ინტერვალი1*, TimeSpan *ინტერვალი2*) (სტატიკური)	int	ადარებს TimeSpan კლასის ორ ობიექტს. გასცემს შესაბამის რიცხვს
Equals(TimeSpan *ინტერვალი1*, TimeSpan *ინტერვალი2*)	bool	ადარებს TimeSpan კლასის ორ ობიექტს. გასცემს შესაბამის ლოგიკურ მნიშვნელობას
FromDays(double *დღეების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს დღეების მითითებულ რაოდენობას
FromHours(double *საათების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს საათების მითითებულ რაოდენობას
FromMilliseconds(double *მილიწამების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს მილიწამების მითითებულ რაოდენობას
FromMinutes(double *წუთების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს წუთების მითითებულ რაოდენობას
FromSeconds(double *წამების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს წამების მითითებულ რაოდენობას
FromTicks(long long *წიკების_რაოდენობა*) (სტატიკური)	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომელიც შეიცავს წიკების მითითებულ რაოდენობას
Parse(String^ *დროის_ინტერვალი*) (სტატიკური)	TimeSpan	დროის ინტერვალის მითითებულ სტრიქონულ წარმოდგენას გარდაქმნის TimeSpan კლასის ეკვივალენტურ ობიექტად
Add(TimeSpan *პარამეტრი*)	TimeSpan	TimeSpan კლასის მოცემულ ობიექტს უმატებს ამავე კლასის ობიექტს
CompareTo(TimeSpan *დროის_ინტერვალი*)	int	ადარებს TimeSpan კლასის ორ ობიექტს. გასცემს შესაბამის რიცხვს
Duration()	TimeSpan	გასცემს დადებითი ხანგრძლივობის TimeSpan კლასის ობიექტს, რომელიც სიდიდით მიმდინარე ობიექტის ხანგრძლივობის ტოლია
Negate()	TimeSpan	გასცემს TimeSpan კლასის ობიექტს, რომლის მნიშვნელობა მიმდინარე ობიექტის მნიშვნელობის საწინააღმდეგოა
Subtract(TimeSpan *დროის_ინტერვალი*)	TimeSpan	TimeSpan კლასის მოცემულ ობიექტს აკლებს ამავე კლასის ობიექტს
ToString()	String^	TimeSpan კლასის მოცემულ ობიექტს გარდაქმნის ეკვივალენტურ სტრიქონად

TotalDays, TotalHours, TotalMinutes, TotalSeconds და **TotalMilliseconds** თვისებებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
//      პროგრამაში ხდება TotalDays, TotalHours, TotalMinutes, TotalSeconds და
//      TotalMilliseconds თვისებებთან მუშაობის დემონსტრირება
TimeSpan Drois_Intervali1(5, 15, 45);
label1->Text  = Drois_Intervali1.TotalDays.ToString() + "\n";
label1->Text += Drois_Intervali1.TotalHours.ToString() + "\n";
label1->Text += Drois_Intervali1.TotalMinutes.ToString() + "\n";
label1->Text += Drois_Intervali1.TotalSeconds.ToString() + "\n";
label1->Text += Drois_Intervali1.TotalMilliseconds.ToString() + "\n";
}
```

FromDays(), FromHours(), FromMinutes(), FromSeconds(), FromMilliseconds() და **FromTicks()** ფუნქციების პარამეტრებს აქვთ double ტიპი. ამ ფუნქციებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
//      FromDays(), FromHours(), FromMinutes(), FromSeconds(),
//      FromMilliseconds() და FromTicks() ფუნქციებთან მუშაობის დემონსტრირება
TimeSpan Drois_Intervali1 = TimeSpan::FromDays(7);
TimeSpan Drois_Intervali2 = TimeSpan::FromHours(7);
TimeSpan Drois_Intervali3 = TimeSpan::FromMinutes(7);
TimeSpan Drois_Intervali4 = TimeSpan::FromSeconds(7);
TimeSpan Drois_Intervali5 = TimeSpan::FromMilliseconds(7);
TimeSpan Drois_Intervali6 = TimeSpan::FromTicks(7);
label1->Text  = Drois_Intervali1.ToString() + "\n";
label1->Text += Drois_Intervali2.ToString() + "\n";
label1->Text += Drois_Intervali3.ToString() + "\n";
label1->Text += Drois_Intervali4.ToString() + "\n";
label1->Text += Drois_Intervali5.ToString() + "\n";
label1->Text += Drois_Intervali6.ToString() + "\n";
}
```

Parse() ფუნქცია. მისთვის პარამეტრად გადაცემულ სტრიქონს უნდა ჰქონდეს შემდეგი ფორმატი:

[ws] [-] [d.]hh:mm:ss[.ff][ws]

კვადრატულ ფრჩხილებში მოთავსებული ელემენტების მითითება აუცილებელი არ არის.

ფორმატის ელემენტები მოცემულია 12.7 ცხრილში.

ცხრილი 12.7. Parse() ფუნქციის ფორმატის ელემენტები

ელემენტი	აღწერა
ws	არააუცილებელი " " (ინტერვალის სიმბოლო)
–	არააუცილებელი "-", რომელიც აღნიშნავს, რომ ინტერვალი უარყოფითია
d	დღეების არააუცილებელი რაოდენობა
hh	საათი ოცდაოთხსაათიან ფორმატში
mm	წუთები
ss	წამები
ff	წამის არააუცილებელი ნაწილები. ის შეიძლება შეიცავდეს 1÷7 სიმბოლოს

მაგალითი:

```
{
//      Parse() ფუნქციის ფორმატის ელემენტებთან მუშაობის დემონსტრირება
//      იქმნება Drois_Intervali1 ობიექტი, რომლის ხანგრძლივობაა 9 საათი, 15 წუთი და 45 წამი
TimeSpan Drois_Intervali1 = TimeSpan::Parse("9:15:45");
//      იქმნება Drois_Intervali2 ობიექტი, რომლის ხანგრძლივობაა
//      2 დღე, 9 საათი, 15 წუთი და 45.50 წამი
TimeSpan Drois_Intervali2 = TimeSpan::Parse("2.9:15:45.50");
label1->Text  = Drois_Intervali1.ToString() + "\n";
label1->Text += Drois_Intervali2.ToString() + "\n";
}
```

Add() და **Subtract()** ფუნქციებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
TimeSpan Drois_Intervali1 (2, 15, 25);
TimeSpan Drois_Intervali2 (12, 20, 30);
TimeSpan Drois_Intervali3 = Drois_Intervali2.Add(Drois_Intervali1);
TimeSpan Drois_Intervali4 = Drois_Intervali2.Subtract(Drois_Intervali1);
label1->Text  = Drois_Intervali3.ToString() + "\n";          //      შედეგი - 14:35:55
label1->Text += Drois_Intervali4.ToString() + "\n";          //      შედეგი - 10:05:05
}
```

Duration() და **Negate()** ფუნქციებთან მუშაობის დემონსტრირება ხდება მოცემული პროგრამით.

```
{
TimeSpan Drois_Intervali1(2, 15, 25);
TimeSpan Drois_Intervali2 = Drois_Intervali1.Duration();
TimeSpan Drois_Intervali3 = Drois_Intervali1.Negate();
label1->Text  = Drois_Intervali2.ToString() + "\n";          //      შედეგი - 02:15:25
label1->Text += Drois_Intervali3.ToString() + "\n";          //      შედეგი - -02:15:25
}
```

თავი 14. კოლექცია

მე-4 თავში ჩვენ განვიხილეთ მასივი. იგი ფართოდ გამოიყენება სხვადასხვა ხასიათის ამოცანის გადაწყვეტად. ამავე დროს, მას აქვს შემდეგი შეზღუდვები:

- შექმნის შემდეგ მასივს ენიჭება ფიქსირებული ზომა, რომელსაც ვეღარ შევცვლით.
- შეუძლებელია მასივში ელემენტის ჩამატება ან წაშლა.
- მასივის ელემენტებთან მიმართვა შესაძლებელია მხოლოდ ინდექსის საშუალებით.
- მასივის ელემენტებს ერთი და იგივე ტიპი აქვთ.

ეს შეზღუდვები მოხსნილია კოლექციაში. ჩვენ განვიხილავთ შემდეგ კოლექციებს: დინამიკური მასივი, ჰეშ-ცხრილი, დახარისხებული სია, რიგი, სტეკი და ბიტების მასივი. კოლექციის აღმწერი კლასები გამოცხადებულია System::Collections სახელების სივრცეში. ამ განყოფილებაში მოცემული ყველა პროგრამის დასაწყისში უნდა მოვათავსოთ დირექტივა:

```
using namespace System::Collections;
```

დინამიკური მასივი

ჩვეულებრივი მასივისაგან განსხვავებით, ***დინამიკური მასივი*** მასში ელემენტების დამატების ან წაშლის შემთხვევაში იცვლის ზომას. დინამიკურ მასივში ელემენტების ნუმერაცია ნულიდან იწყება. ArrayList კლასში განსაზღვრულია თვისებები და ფუნქციები დინამიკური მასივის ელემენტებთან სამუშაოდ. ზოგიერთი მათგანი მოცემულია 13.1 და 13.2 ცხრილებში.

დინამიკური მასივის შექმნა. ArrayList კონსტრუქტორს სამი გადატვირთული ვერსია აქვს. შესაბამისად, დინამიკური სტრიქონი სამი გზით შეგვიძლია შევქმნათ. ამ ვერსიების სინტაქსია:

ArrayList()
ArrayList(ICollection^ *კოლექცია*)
ArrayList(int *ტევადობა*)

მაგალითი:

```
{
array<String^>^ Striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
array<int>^ masivi = gcnew array<int> { 1, 2, 3, 4, 5 };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList();
ArrayList^ DinamiuriMasivi2 = gcnew ArrayList(masivi);
ArrayList^ DinamiuriMasivi3 = gcnew ArrayList(Striqoni);
ArrayList^ DinamiuriMasivi4 = gcnew ArrayList(50);

for each (int x in DinamiuriMasivi2) label1->Text += x.ToString() + " ";
for each (String^ x in DinamiuriMasivi3) label2->Text += x + " ";
}
```

ცხრილი 13.1. ArrayList კლასის თვისებები

თვისება	ტიპი	აღწერა
Capacity	int	განსაზღვრავს დინამიკური მასივის ტევადობას, ანუ ელემენტების მაქსიმალურ რაოდენობას
Count	int	შეიცავს დინამიკურ მასივში ელემენტების რაოდენობას
IsFixedSize	bool	მიუთითებს, არის თუ არა დინამიკური მასივი ფიქსირებული ზომის
IsReadOnly	bool	მიუთითებს, დინამიკური მასივი მხოლოდ წაკითხვადია თუ არა
Item	Object^	გასცემს ან აყენებს მითითებული ინდექსის მქონე ელემენტის მნიშვნელობას

ცხრილი 13.2. ArrayList კლასის ფუნქციები

ფუნქცია	დაბრუნებული მნიშვნელობის ტიპი	აღწერა
ReadOnly(ArrayList^ *მასივი*)	ArrayList^	გასცემს დინამიკურ მასივს მხოლოდ წაკითხვისათვის
Repeat(Object^ *სიდიდე*, int რაოდენობა)	ArrayList^	გასცემს დინამიკური მასივს, რომლის ყველა ელემენტი გადაცემული მნიშვნელობების ტოლია
Add(Object^ *სიდიდე*)	int	დინამიკურ მასივს ბოლოში უმატებს ელემენტს
AddRange(ICollection^ *კოლექცია*)	void	დინამიკურ მასივს ბოლოში უმატებს სხვა კოლექციის ელემენტებს
BinarySearch(Object^ *სიდიდე*)	int	დახარისხებულ მასივში ასრულებს მითითებული ელემენტის ორობით ძებნას.
Clear()	void	დინამიკური მასივიდან შლის ყველა ელემენტს
Contains(Object^ *სიდიდე*)	bool	განსაზღვრავს შეიცავს თუ არა დინამიკური მასივი მითითებულ ელემენტს
CopyTo(Array^ *მასივი*)	void	დინამიკური მასივის ელემენტებს გადაწერს ერთგანზომილებიან მასივში
Equals(Object^ *სიდიდე*)	bool	განსაზღვრავს ტოლია თუ არა ორი დინამიკური მასივი
GetEnumerator()	IEnumerator^	გასცემს ჩამომთვლელს
GetRange(int *ინდექსი*, int *რაოდენობა*)	ArrayList^	გასცემს დინამიკური მასივს, რომელიც შეიცავს საწყისი მასივის ელემენტების დიაპაზონს
GetType()	Type^	გასცემს მიმდინარე ობიექტის ტიპს
IndexOf(Object^ *სიდიდე*)	int	გასცემს ნაპოვნი ელემენტის პოზიციის ნომერს
Insert(int *ინდექსი*, Object^ *სიდიდე*)	void	დინამიკურ მასივში ელემენტს ჩასვამს მითითებული პოზიციიდან
InsertRange(int *ინდექსი*, ICollection^ *კოლექცია*)	void	დინამიკურ მასივში ჩასვამს კოლექციის ელემენტს მითითებული პოზიციიდან
LastIndexOf(Object^ *სიდიდე*)	int	გასცემს ნაპოვნი ელემენტის ბოლო პოზიციის ნომერს

ცხრილი 13.2 (გაგრძელება)

Remove(Object^ *სიდიდე*)	void	დინამიკური მასივიდან შლის პირველ ნაპოვნ ელემენტს
RemoveAt(int *ინდექსი*)	void	დინამიკური მასივიდან შლის მითითებული პოზიციის ელემენტს
RemoveRange(int *ინდექსი*, int *რაოდენობა*)	void	დინამიკური მასივიდან შლის ელემენტების დიაპაზონს დაწყებული მითითებული პოზიციიდან
Reverse()	void	დინამიკური მასივის ელემენტებს უკუმიმდევ-რობით ალაგებს
SetRange(int *ინდექსი*, ICollection^ *კოლექცია*)	void	დინამიკური მასივის შესაბამის დიაპაზონში ჩასვამს კოლექციის ელემენტებს
Sort()	void	ახარისხებს დინამიკურ მასივს ან მის დიაპაზონს
ToArray()	array<Object^>^	დინამიკური მასივის ელემენტებს გადაწერს ჩვეულებრივ მასივში
ToString()	String^	მიმდინარე ობიექტს გარდაქმნის სტრიქონად
StrimToSize()	void	ამცირებს დინამიკური მასივის ტევადობას ელემენტების იმ რაოდენობამდე, რომელიც აქვს ამ მასივს ფუნქციის გამოძახების მომენტში

ელემენტის დამატება და ჩამატება. Add() ფუნქცია დინამიკურ მასივს ბოლოში უმატებს მითითებულ ელემენტს, რომლის ტიპია object. მისი სინტაქსია:

arraList*.Add*(სიდიდე)

აქ *arrayList* აღნიშნავს ArrayList ტიპის ობიექტს.

მოცემული პროგრამით ხდება Add ფუნქციასთან მუშაობის დემონსტრირება.

```
{
//      პროგრამაში ხდება Add ფუნქციასთან მუშაობის დემონსტრირება
ArrayList^ DinamiuriMasivi = gcnew ArrayList();
//
DinamiuriMasivi->Add(L"საბა");
DinamiuriMasivi->Add(L"ანა");
DinamiuriMasivi->Add(L"ლიკა");
DinamiuriMasivi->Add(L"რომანი");
//
for ( int indexi = 0; indexi < DinamiuriMasivi->Count; indexi++ )
        label1->Text += L"DinamiuriMasivi [" + indexi.ToString() + "] = " +
        DinamiuriMasivi[indexi]->ToString() + "\n";
}
```

დინამიკურ მასივს შეგვიძლია დავუმატოთ სხვადასხვა ტიპის მონაცემი:

```
{
//      დინამიკურ მასივს სხვადასხვა ტიპის მონაცემი ემატება
ArrayList^ DinamiuriMasivi = gcnew ArrayList();
//
DinamiuriMasivi->Add(1);
DinamiuriMasivi->Add(2.2);
DinamiuriMasivi->Add(true);
```

```
DinamiuriMasivi->Add(L"რომანი");
for ( int indexi = 0; indexi < DinamiuriMasivi->Count; indexi++ )
   label1->Text += L"DinamiuriMasivi[" + indexi.ToString() + "] = " +
                        DinamiuriMasivi[indexi]->ToString() + "\n";
}
```

AddRange() ფუნქცია დინამიკურ მასივს ბოლოში უმატებს ელემენტების დიაპაზონს. მისი სინტაქსია:

***arrayList*.Insert(*კოლექცია*)**

ფუნქციის შესულების შედეგად კოლექცია დაემატება arrayList დინამიკურ მასივს.

მაგალითი:

```
{
array<String^>^ striqoni1 = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი", L"გიორგი" };
array<String^>^ striqoni2 = gcnew array<String^> { L"ნატა", L"ბექა" };

ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni1);
DinamiuriMasivi1->AddRange(striqoni2);      // DinamiuriMasivi1 = "ანა", "საბა", "ლიკა",
                                            // "რომანი", "გიორგი", "ნატა", "ბექა"
for ( int index = 0; index < DinamiuriMasivi1->Count; index++ )
        label1->Text += DinamiuriMasivi1[index] + "  ";
}
```

Insert() ფუნქცია მითითებულ ელემენტს ჩასვამს მითითებულ პოზიციაში. მისი სინტაქსია:

***arrayList*.Insert(*ინდექსი, სიდიდე*)**

მაგალითი:

```
{
array<String^>^ striqoni1 = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი", L"გიორგი" };

ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni1);
DinamiuriMasivi1->Insert(2, L"ლევანი");      // DinamiuriMasivi1 = "ანა", "საბა",
                                             // "ლევანი", "ლიკა", "რომანი", "გიორგი"
for ( int index = 0; index < DinamiuriMasivi1->Count; index++ )
        label1->Text += DinamiuriMasivi1[index] + "  ";
}
```

InsertRange() ფუნქცია კოლექციას ამატებს მითითებული პოზიციიდან. მისი სინტაქსია:

***arrayList*.InsertRange(*ინდექსი, კოლექცია*)**

მაგალითი:

```
{
array<String^>^ striqoni1 = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი", L"გიორგი" };
array<String^>^ striqoni2 = gcnew array<String^> { L"ნატა", L"ბექა" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni1);
DinamiuriMasivi1->InsertRange(2, striqoni2);          // DinamiuriMasivi1 = "ანა", "საბა",
                                                      // "ნატა", "ბექა", "ლიკა", "რომანი", "გიორგი"
for ( int index = 0; index < DinamiuriMasivi1->Count; index++ )
        label1->Text += DinamiuriMasivi1[index] + "  ";
}
```

SetRange() ფუნქცია ცვლის დინამიკური მასივის ელემენტების დიაპაზონს კოლექციის ელემენტებით დაწყებული მითითებული ინდექსიდან. მისი სინტაქსია:

***arrayList*.SetRange(*ინდექსი, კოლექცია*)**

მაგალითი:

```
{
array<String^>^ striqoni1 = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი", L"გიორგი" };
array<String^>^ striqoni2 = gcnew array<String^> { L"ნატა", L"ბექა" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni1);
DinamiuriMasivi1->SetRange(2, striqoni2);                    //    DinamiuriMasivi1 = "ანა", "საბა",
                                                             //    "ნატა", "ბექა", "გიორგი"
for ( int index = 0; index < DinamiuriMasivi1->Count; index++ )
    label1->Text += DinamiuriMasivi1[index] + "  ";
}
```

ელემენტის ძებნა. Contains() ფუნქცია ამოწმებს შეიცავს თუ არა დინამიკური მასივი მითითებულ ელემენტს. თუ ელემენტი სიაშია, მაშინ ფუნქცია გასცემს true მნიშვნელობას, წინააღმდეგ შემთხვევაში კი - false მნიშვნელობას. მისი სინტაქსია:

***arrayList*.Contains(*სიდიდე*)**

მაგალითი:

```
{
bool elementi_aris;
array<String^>^ striqoni = gcnew array<String^>
                                    { L"რომანი", L"ანა", L"საბა", L"ლიკა", L" რომანი ", L"საბა" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
elementi_aris = DinamiuriMasivi1->Contains(L"ანა");
if ( elementi_aris == true ) label1->Text = L"ელემენტი მასივში არის";
    else label1->Text = L"ელემენტი მასივში არ არის";
}
```

IndexOf() ფუნქცია დინამიკურ მასივში ეძებს მითითებულ ელემენტს. თუ ეს ელემენტი რამდენიმეა მასივში, მაშინ გაიცემა პირველი მათგანის ინდექსი, წინააღმდეგ შემთხვევაში კი - -1. მისი სინტაქსია:

***arrayList*.IndexOf(*სიდიდე*)**

მაგალითი:

```
{
int index1;
array<String^>^ striqoni = gcnew array<String^>
                                    { L"რომანი", L"ანა", L"საბა", L"ლიკა", L"რომანი", L"საბა" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
index1 = DinamiuriMasivi1->IndexOf(L"საბა");                    //    index1 = 2
label1->Text =  L" პირველი " + DinamiuriMasivi1[index1] +
            L" ელემენტის ინდექსია " + index1.ToString();
}
```

LastIndexOf() ფუნქცია დინამიკურ მასივში ეძებს მითითებულ ელემენტს. თუ ეს ელემენტი რამდენიმეა მასივში, მაშინ გაიცემა უკანასკნელი მათგანის ინდექსი, წინააღმდეგ შემთხვევაში კი - -1. მისი სინტაქსია:

***arrayList*.LastIndexOf(*სიდიდე*)**

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^>
                                    { L"რომანი", L"ანა", L"საბა", L"ლიკა", L"რომანი", L"საბა" };
```

```
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
index1 = DinamiuriMasivi1->LastIndexOf(L"რომანი");                    // index1 = 4
label1->Text = L" უკანასკნელი " + DinamiuriMasivi1[index1] +
                L" ელემენტის ინდექსია " + index1.ToString();
}
```

ელემენტების წაშლა. RemoveAt() ფუნქცია. ის დინამიკური მასივიდან შლის მითითებული ინდექსის მქონე ელემენტს. მისი სინტაქსია:

***arrayList*.RemoveAt(*ინდექსი*)**

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
DinamiuriMasivi1->RemoveAt(1);              // DinamiuriMasivi1 = "ანა", "ლიკა", "რომანი"
for each ( String^ str1 in DinamiuriMasivi1 ) label1->Text += str1 + "  ";
}
```

Remove() ფუნქცია. ის დინამიკური მასივიდან შლის მითითებულ ელემენტს. მისი სინტაქსია:

***arrayList*.Remove(*სიდიდე*)**

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
DinamiuriMasivi1->Remove(L"რომანი");      // DinamiuriMasivi1 = "ანა", "საბა", "ლიკა"
for each ( String^ str1 in DinamiuriMasivi1 ) label1->Text += str1 + "  ";
}
```

RemoveRange() ფუნქცია. ის დინამიკური მასივიდან შლის ელემენტების დიაპაზონს. მისი სინტაქსია:

***arrayList*.RemoveRange(*საწყისი_ინდექსი, ელემენტების_რაოდენობა*)**

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
DinamiuriMasivi1->RemoveRange(1, 2);          // DinamiuriMasivi1 = "ანა", "რომანი"
for each ( String^ str1 in DinamiuriMasivi1 ) label1->Text += str1 + "  ";
}
```

ელემენტების დახარისხება, ძებნა და მიმდევრობის შებრუნება. Sort() ფუნქცია დინამიკური მასივის ელემენტებს ზრდადობის მიხედვით ალაგებს. მისი სინტაქსია:

***arrayList*.Sort()**

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
DinamiuriMasivi1->Sort();      // DinamiuriMasivi1 = "ანა", "ლიკა", "რომანი", "საბა"
for each ( String^ str1 in DinamiuriMasivi1 ) label1->Text += str1 + "  ";
}
```

BinarySearch() ფუნქცია დახარისხებულ დინამიკურ მასივში ეძებს მითითებულ ელემენტს. პოვნის შემთხვევაში გაიცემა მისი ინდექსი. მისი სინტაქსია:

arrayList.BinarySearch(*ინდექსი*)

მაგალითი:

```
{
int indexi;
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
DinamiuriMasivi1->Sort();
indexi = DinamiuriMasivi1->BinarySearch(L"რომანი");                    //        indexi = 2
label1->Text = indexi.ToString();
}
```

Reverse() ფუნქცია ახდენს დინამიკური მასივის ელემენტების მიმდევრობის შებრუნებას. მისი სინტაქსია:

arrayList.Reverce()

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"ანა", L"საბა", L"ლიკა", L"რომანი" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
for each ( String^ str1 in DinamiuriMasivi1 ) label1->Text += str1 + "  ";
DinamiuriMasivi1->Reverse();          // DinamiuriMasivi1 = "ანა", "რომანი", "ლიკა", "საბა"
for each ( String^ str1 in DinamiuriMasivi1 ) label2->Text += str1 + "  ";
}
```

დინამიკური მასივის ტევადობის შემცირება და ელემენტების დიაპაზონის მიღება. TrimToSize() ფუნქცია დინამიკური მასივის ტევადობას ამცირებს მასივში რეალურად არსებული ელემენტების რაოდენობამდე. მისი სინტაქსია:

arrayList.TrimToSize()

GetRange() ფუნქცია გასცემს დინამიკური მასივის მითითებული რაოდენობის ელემენტს დაწყებული მითითებული პოზიციიდან. მისი სინტაქსია:

arrayList.GetRange(*საწყისი_ინდექსი, ელემენტების_რაოდენობა*)

მაგალითი:

```
{
array<String^>^ striqoni = gcnew array<String^> { L"საბა", L"ლიკა", L"რომანი", L"ანა", "ნატა" };
ArrayList^ DinamiuriMasivi1 = gcnew ArrayList(striqoni);
ArrayList^ DinamiuriMasivi2 = DinamiuriMasivi1->GetRange(1, 3);
                                                    // DinamiuriMasivi2 = "ლიკა", "რომანი", "ანა"
for each ( String^ str1 in DinamiuriMasivi2 ) label1->Text += str1 + "  ";
}
```

ჩამომთვლელების გამოყენება. ვნახოთ თუ როგორ შეიძლება ჩამომთვლელების გამოყენება დინამიკური მასივიდან ელემენტების წაკითხვისათვის. ზემოთ აღნიშნული ყველა სახის კოლექცია ახდენს IEnumerable ინტერფეისის რეალიზებას. ამ ინტერფეისში განსაზღვრულია Current თვისება და GetEnumerator(), MoveNext() და Reset() ფუნქციები.

GetEnumerator() ფუნქცია გასცემს ჩამომთვლელს, რომელიც IEnumerator ობიექტია და გამოიყენება კოლექციის ელემენტებთან მიმართვისათვის. ჩვენ განვიხილავთ მისი ორი ვერსიის სინტაქსს:

arrayList.GetEnumerator()

arrayList.GetEnumerator(*საწყისი_ინდექსი, ელემენტების_რაოდენობა*)

MoveNext() ფუნქცია ჩამომთვლელს ათავსებს კოლექციის მომდევნო ელემენტზე და გასცემს true მნიშვნელობას თუ მომდევნო ელემენტი არსებობს, წინააღმდეგ შემთხვევაში კი -

false მნიშვნელობას.

Reset() ფუნქცია ჩამომთვლელს ათავსებს საწყის მდგომარეობაში ანუ კოლექციის პირველი ელემენტის წინ.

მოცემული პროგრამით ხდება დინამიკურ მასივთან ჩამომთვლელის საშუალებით მუშაობის დემონსტრირება.

```
{
//      დინამიკურ მასივთან ჩამომთვლელის საშუალებით
//      მუშაობის დემონსტრირება
array<String^>^ striqoni = gcnew array<String^> { L"საბა", L"ლიკა", L"რომანი", L"ანა", L"ნატა" };
ArrayList^ DinamiuriMasivi = gcnew ArrayList(striqoni);
IEnumerator^ Chamomtvleli = DinamiuriMasivi->GetEnumerator();
while ( Chamomtvleli->MoveNext() )
label1->Text += L"myEnumerator.Current = " + Chamomtvleli->Current->ToString() + "\n";
//
Chamomtvleli->Reset();
Chamomtvleli->MoveNext();
label1->Text += L"myEnumerator.Current = " + Chamomtvleli->Current->ToString() + "\n";
}
```

ჰეშ-ცხრილი

ჰეშ-ცხრილი ინახავს წყვილებს - გასაღები-მნიშვნელობა ანუ ჰეშ-ცხრილის თითოეული ელემენტი შედგება გასაღებისა და მნიშვნელობისაგან. გასაღები გამოიყენება შესაბამისი მნიშვნელობის მისაღებად. ჰეშ-ცხრილი განსაზღვრულია Hashtable კლასში. ამ კლასის თვისებები და ფუნქციები მოცემულია 13.3 და 13.4 ცხრილებში. ჰეშ-ცხრილში გასაღები და მნიშვნელობა შეიძლება იყოს ნებისმიერი ტიპის ობიექტი. თუმცა, ჩვეულებრივ, გასაღების როლში ხშირად სტრიქონებს იყენებენ.

ცხრილი 13.3. Hashtable კლასის თვისებები

თვისებები	ტიპი	აღწერა
Count	int	შეიცავს ჰეშ-ცხრილში შენახული ელემენტების რაოდენობას
IsFixedSize	bool	ამოწმებს აქვს თუ არა ჰეშ-ცხრილს ფიქსირებული სიგრძე
IsReadOnly	bool	ამოწმებს ჰეშ-ცხრილი მხოლოდ წაკითხვადია თუ არა
Item	Object^	გასცემს ან აყენებს მითითებული გასაღების მქონე ელემენტის მნიშვნელობას
Keys	ICollection^	გასცემს კოლექციას, რომელიც ჰეშ-ცხრილის გასაღებებისაგან შედგება
Values	ICollection^	გასცემს კოლექციას, რომელიც ჰეშ-ცხრილის მნიშვნელობებისაგან შედგება

ელემენტის დამატება, წაშლა და მიღება. Add() ფუნქცია გამოიყენება ჰეშ-ცხრილში ელემენტების დასამატებლად. მისი სინტაქსია:

***Hashtable*.Add(*გასაღები, მნიშვნელობა*)**

Remove() ფუნქციას იყენებენ ჰეშ-ცხრილში ელემენტების წასაშლელად. მისი სინტაქსია:

Hashtable.Remove(*გასაღები*)

მოცემული პროგრამით ხდება ამ ფუნქციებთან მუშაობის დემონსტრირება.

ცხრილი 13.4. Hashtable კლასის ფუნქციები

ფუნქციები	დაბრუნებული ტიპი	აღწერა
Add(Object^ *გასაღები*, Object^ *სიდიდე*)	void	ჰეშ-ცხრილს უმატებს მითითებული გასაღებისა და მნიშვნელობის ელემენტს
Clear()	void	ჰეშ-ცხრილიდან შლის ყველა ელემენტს
Clone()	Object^	ქმნის ჰეშ-ცხრილის ასლს
Contains(Object^ *გასაღები*)	bool	განსაზღვრავს, შეიცავს თუ არა ჰეშ-ცხრილი მითითებულ გასაღებს
ContainsKey(Object^ *გასაღები*)	bool	განსაზღვრავს, შეიცავს თუ არა ჰეშ-ცხრილი მითითებულ გასაღებს
ContainsValue(Object^ *სიდიდე*)	bool	განსაზღვრავს, შეიცავს თუ არა ჰეშ-ცხრილი მითითებულ მნიშვნელობას
CopyTo(Array^ *მასივი*, int *ინდექსი*)	void	ერთგანზომილებიან მასივში ჰეშ-ცხრილიდან გადაწერს გასაღებებს ან მნიშვნელობებს
Equals(Object^ *სიდიდე*)	bool	ამოწმებს, ტოლია თუ არა ორი ჰეშ-ცხრილი
GetEnumerator()	IDictionary-Enumerator^	გასცემს ჩამომთვლელს, რომელიც გამოიყენება ჰეშ-ცხრილის ელემენტებთან სამუშაოდ
GetType()	Type^	გასცემს მიმდინარე ობიექტის ტიპს
Remove(Object^ *გასაღები*)	void	ჰეშ-ცხრილიდან შლის მითითებული გასაღების მქონე ელემენტს
ToString()	String^	გასცემს მიმდინარე ობიექტის შესაბამის სტრიქონს

```
{
//      პროგრამაში ხდება Add() და Remove() ფუნქციებთან მუშაობის დემონსტრირება
Hashtable^ HeshCxrili = gcnew Hashtable();
//
HeshCxrili->Add(L"კა", L"კახეთი");
HeshCxrili->Add(L"იმ", L"იმერეთი");
HeshCxrili->Add(L"თუ", L"თუშეთი");
HeshCxrili->Add(L"ხე", L"ხევსურეთი");
for each ( String^ Gasagebi in HeshCxrili->Keys )
        label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in HeshCxrili->Values )
        label1->Text += Mnishvneloba->ToString() + "\n";
//
HeshCxrili->Remove(L"თუ");
for each ( String^ Gasagebi in HeshCxrili->Keys )
        label2->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in HeshCxrili->Values )
```

```
        label2->Text += Mnishvneloba->ToString() + "\n";
String^ Regioni = (String^)HeshCxrili[L"იმ"];
label3->Text = Regioni;
}
```

როგორც პროგრამიდან ჩანს, ჰეშ-ცხრილიდან საჭირო მნიშვნელობის მისაღებად კვადრატულ ფრჩხილებში უნდა მივუთითოთ მისი შესაბამისი გასაღები, მაგალითად, HeshCxrili["იმ"]. გასაღების მისაღებად უნდა გამოვიყენოთ Keys თვისება, მნიშვნელობის მისაღებად კი - Values თვისება.

გასაღებისა და მნიშვნელობის ძებნა. ContainsKey() ფუნქცია გამოიყენება ჰეშ-ცხრილში მითითებული გასაღების მოსაძებნად. პოვნის შემთხვევაში გაიცემა true, წინააღმდეგ შემთხვევაში - false. მისი სინტაქსია:

***Hashtable*.ContainsKey(*გასაღები*)**

ContainsValue() ფუნქცია გამოიყენება ჰეშ-ცხრილში მითითებული მნიშვნელობის მოსაძებნად. პოვნის შემთხვევაში გაიცემა true, წინააღმდეგ შემთხვევაში - false. მისი სინტაქსია:

***Hashtable*.ContainsValue(*მნიშვნელობა*)**

მოცემული პროგრამით ხდება ამ ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      ContainsKey() და ContainsValue() ფუნქციებთან მუშაობის დემონსტრირება
 Hashtable^ HeshCxrili = gcnew Hashtable();
//
HeshCxrili->Add(L"კა", L"კახეთი");
HeshCxrili->Add(L"იმ", L"იმერეთი");
HeshCxrili->Add(L"თუ", L"თუშეთი");
HeshCxrili->Add(L"ხე", L"ხევსურეთი");
for each ( String^ Gasagebi in HeshCxrili->Keys )
      label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in HeshCxrili->Values )
      label1->Text += Mnishvneloba->ToString() + "\n";
//
if ( HeshCxrili->ContainsKey(L"კა") )
        label2->Text = L"ჰეშ-ცხრილი შეიცავს მითითებულ გასაღებს";
else    label2->Text = L"ჰეშ-ცხრილი არ შეიცავს მითითებულ გასაღებს";
//
if ( HeshCxrili->ContainsValue(L"ხევსურეთი") )
        label3->Text = L"ჰეშ-ცხრილი შეიცავს მითითებულ მნიშვნელობას";
else    label3->Text = L"ჰეშ-ცხრილი არ შეიცავს მითითებულ მნიშვნელობას";
}
```

გასაღებებისა და მნიშვნელობების გადაწერა ერთგანზომილებიან მასივში. CopyTo() ფუნქცია გამოიყენება ჰეშ-ცხრილიდან გასაღებებისა და მნიშვნელობების გადასაწერად ერთგანზომილებიან მასივში. მისი სინტაქსია:

***Hashtable*.Keys.CopyTo(*მასივი, ინდექსი*)**

ფუნქცია დაწყებული ინდექსი პოზიციიდან მითითებულ მასივში გადაწერს ჰეშ-ცხრილის გასაღებებს.

***Hashtable*.Values.CopyTo(*მასივი, ინდექსი*)**

ფუნქცია დაწყებული ინდექსი პოზიციიდან მითითებულ მასივში გადაწერს ჰეშ-ცხრილის მნიშვნელობებს.

მოცემული პროგრამით ხდება ამ ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      CopyTo() ფუნქციასთან მუშაობის დემონსტრირება
Hashtable^ HeshCxrili = gcnew Hashtable();
HeshCxrili->Add(L"კა", L"კახეთი");
HeshCxrili->Add(L"იმ", L"იმერეთი");
HeshCxrili->Add(L"თუ", L"თუშეთი");
HeshCxrili->Add(L"ხე", L"ხევსურეთი");
array<String^>^ Gasagebebi = gcnew array<String^> (HeshCxrili->Count);
array<String^>^ Mnishvnelobebi = gcnew array<String^> (HeshCxrili->Count);

for each ( String^ Gasagebi in HeshCxrili->Keys )
        label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in HeshCxrili->Values )
        label1->Text += Mnishvneloba->ToString() + "\n";
//
HeshCxrili->Keys->CopyTo(Gasagebebi, 0);
for each ( String^ Gasagebi in Gasagebebi )
        label2->Text += Gasagebi + "\n";
//
HeshCxrili->Values->CopyTo(Mnishvnelobebi, 0);
for each ( String^ Mnishvneloba in Mnishvnelobebi )
        label3->Text += Mnishvneloba + "\n";
}
```

დახარისხებული სია

დახარისხებული სია არის დინამიკური მასივისა და ჰეშ-ცხრილის კომბინაცია. დახარისხებული სიის ელემენტის მიღება შეიძლება როგორც ინდექსის, ისე გასაღების მიხედვით. გასაღებები ასეთ მასივში დახარისხებულია ზრდადობის მიხედვით. მასივში ახალი ელემენტის ჩამატებისას ის იმ პოზიციაში მოთავსდება, რომ მასივი დახარისხებული დარჩეს. დახარისხებული სია განსაზღვრულია SortedList კლასში. ამ კლასის თვისებები და ფუნქციები მოცემულია 13.5 და 13.6 ცხრილებში.

ელემენტების დამატება, წაშლა და მიღება. Add() ფუნქცია გამოიყენება დახარისხებულ სიაში ელემენტების დასამატებლად. მისი სინტაქსია:

***sortedList*.Add(*გასაღები, მნიშვნელობა*)**

Remove() ფუნქცია გამოიყენება დახარისხებული სიიდან ელემენტების წასაშლელად. მისი სინტაქსია:

***sortedList*.Remove(*გასაღები*)**

ცხრილი 13.5. SortedList კლასის თვისებები

თვისება	ტიპი	აღწერა
Capacity	int	განსაზღვრავს დახარისხებული სიის ტევადობას
Count	int	შეიცავს დახარისხებულ სიაში რეალურად არსებული ელემენტების რაოდენობას
IsFixedSize	bool	ამოწმებს, აქვს თუ არა დახარისხებულ მასივს ფიქსირებული სიგრძე
IsReadOnly	bool	ამოწმებს, არის თუ არა დახარისხებული სია მხოლოდ წაკითხვადი
Item	Object^	გასცემს ან აყენებს მითითებული გასაღების მქონე ელემენტის მნიშვნელობას
Keys	ICollection^	გასცემს დახარისხებული სიის გასაღებებისაგან შემდგარ კოლექციას
Values	ICollection^	გასცემს დახარისხებული სიის მნიშვნელობებისაგან შემდგარ კოლექციას

ცხრილი 13.6. SortedList კლასის ფუნქციები

ფუნქცია	დაბრუნებული ტიპი	აღწერა
Add(Object^ *გასაღები*, Object^ *სიდიდე*)	void	დახარისხებულ სიას ელემენტს უმატებს
Clear()	void	დახარისხებული სიიდან შლის ყველა ელემენტს
Clone()	Object^	ქმნის დახარისხებული სიის ასლს
Contains(Object^ *გასაღები*)	bool	მითითებულ გასაღებს ეძებს დახარისხებულ სიაში
ContainsKey(Object^ *გასაღები*)	bool	მითითებულ გასაღებს ეძებს დახარისხებულ სიაში
ContainsValue(Object^ *სიდიდე*)	bool	მითითებულ მნიშვნელობას ეძებს დახარისხებულ სიაში
CopyTo(Array^ *მასივი*, int *ინდექსი*)	void	ერთგანზომილებიან მასივში გადაწერს დახარისხებული სიის მნიშვნელობებს ან გასაღებებს
Equals(Object^ *ობიექტი*)	bool	ამოწმებს, ტოლია თუ არა ორი ობიექტი
GetByIndex(int *ინდექსი*)	Object^	დახარისხებული სიიდან გასცემს მითითებული ინდექსის მქონე მნიშვნელობას
GetEnumerator()	IDictionaryEnumerator^	გასცემს ჩამომთვლელს დახარისხებული სიისათვის
GetKey(int *ინდექსი*)	Object^	გასცემს დახარისხებული სიის მითითებული პოზიციის გასაღებს
GetKeyList()	IList^	გასცემს დახარისხებული სიის ყველა გასაღებს
GetType()	Type^	გასცემს მოცემული ობიექტის ტიპს
GetValueList()	IList^	გასცემს დახარისხებული სიის ყველა მნიშვნელობას

ცხრილი 13.6. (გაგრძელება)

IndexOfKey(Object^ *გასაღები*)	int	გასცემს მითითებული გასაღების ინდექსს
IndexOfValue(Object^ *სიდიდე*)	int	გასცემს მითითებული მნიშვნელობის ინდექსს
Remove(Object^ *გასაღები*)	void	დახარისხებული სიიდან შლის მითითებული გასაღების შემცველ ელემენტს
RemoveAt(int *ინდექსი*)	void	დახარისხებული სიიდან შლის მითითებული მნიშვნელობის შემცველ ელემენტს
SetByIndex(int *ინდექსი*, Object^ *სიდიდე*)	void	ცვლის მნიშვნელობას ინდექსით მითითებულ პოზიციაში
ToString()	String^	გასცემს მიმდინარე ობიექტის შესაბამის სტრიქონს
TrimToSize()	void	დახარისხებული სიის ზომას ამცირებს მასში რეალურად არსებული ელემენტების რაოდენობამდე

მოცემული პროგრამით ხდება ამ ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      პროგრამაში ხდება Add() და Remove() ფუნქციებთან მუშაობის დემონსტრირება
SortedList^ DaxarisxebuliSia = gcnew SortedList ();
//
DaxarisxebuliSia->Add(L"კა", L"კახეთი");
DaxarisxebuliSia->Add(L"იმ", L"იმერეთი");
DaxarisxebuliSia->Add(L"თუ", L"თუშეთი");
DaxarisxebuliSia->Add(L"ხე", L"ხევსურეთი");
for each ( String^ Gasagebi in DaxarisxebuliSia->Keys )
        label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ MniSvneloba in DaxarisxebuliSia->Values )
label1->Text += MniSvneloba->ToString() + "\n";
//
DaxarisxebuliSia->Remove(L"თუ");
for each ( String^ Gasagebi in DaxarisxebuliSia->Keys )
        label2->Text += Gasagebi->ToString() + "\n";
for each ( String^ MniSvneloba in DaxarisxebuliSia->Values )
        label2->Text += MniSvneloba->ToString() + "\n";
String^ Regioni = (String^)DaxarisxebuliSia[L"იმ"];
label3->Text += Regioni;
}
```

როგორც პროგრამიდან ჩანს, დახარისხებული სიიდან საჭირო მნიშვნელობის მისაღებად კვადრატულ ფრჩხილებში უნდა მივუთითოთ მისი შესაბამისი გასაღები, მაგალითად, DaxarisxebuliSia["იმ"]. გასაღებების მისაღებად უნდა გამოვიყენოთ Keys თვისება, მნიშვნელობების მისაღებად კი - Values თვისება.

მნიშვნელობის მიღება და შეცვლა ინდექსის მიხედვით. GetByIndex() ფუნქცია გამოიყენება დახარისხებული სიიდან საჭირო მნიშვნელობის მისაღებად ინდექსის მიხედვით. მისი სინტაქსია:

***sortedList*.GetByIndex(*ინდექსი*)**

მაგალითი:

```
String^ regioni1 = ( String^ ) DaxarisxebuliSia->GetByIndex(2);
```

SetByIndex() ფუნქცია გამოიყენება დახარისხებულ სიაში მნიშვნელობის შესაცვლელად ინდექსის მიხედვით. მისი სინტაქსია:

***sortedList*.SetByIndex(*ინდექსი, მნიშვნელობა*)**

მაგალითი:

```
DaxarisxebuliSia->SetByIndex(2, L"რაჭა");
```

გასაღების, გასაღებისა და მნიშვნელობის ინდექსის მიღება. GetKey() ფუნქცია გამოიყენება დახარისხებული სიიდან გასაღების მისაღებად ინდექსის მიხედვით. მისი სინტაქსია:

***sortedList*.GetKey(*ინდექსი*)**

მაგალითი:

```
String^ Gasagebi = ( String^ ) DaxarisxebuliSia->GetKey(2);
```

IndexOfKey() ფუნქცია გამოიყენება დახარისხებული სიიდან მითითებული გასაღების ინდექსის მისაღებად. მისი სინტაქსია:

***sortedList*.IndexOfKey(*გასაღები*)**

მაგალითი:

```
int GasagebisIndeqsi = DaxarisxebuliSia->IndexOfKey(L"იმ");
```

IndexOfValue() ფუნქცია გამოიყენება დახარისხებული სიიდან მითითებული მნიშვნელობის ინდექსის მისაღებად. მისი სინტაქსია:

***sortedList*.IndexOfValue(*მნიშვნელობა*)**

მაგალითი:

```
int MnishvnelobisIndeqsi = DaxarisxebuliSia->IndexOfValue(L"იმერეთი");
```

გასაღებებისა და მნიშვნელობების სიის მიღება. GetKeyList() ფუნქცია გამოიყენება დახარისხებული სიიდან გასაღებების სიის გასაცემად. ფუნქცია გასცემს IList ინტერფეისის ობიექტს. ფუნქციის სინტაქსია:

***sortedList*.GetKeyList()**

მაგალითი:

```
{
IList^ GasagebebisSia = DaxarisxebuliSia->GetKeyList();
for each ( String^ Gasagebi in GasagebebisSia )
        label4->Text += Gasagebi + "   ";
}
```

GetValueList() ფუნქცია გამოიყენება დახარისხებული სიიდან მნიშვნელობების სიის გასაცემად. ფუნქცია გასცემს IList ინტერფეისის ობიექტს. ფუნქციის სინტაქსია:

***sortedList*. GetValueList()**

მაგალითი:

```
{
IList^ MnishvnelobebisSia = DaxarisxebuliSia->GetValueList();
for each ( String^ Gasagebi in MnishvnelobebisSia )
        label4->Text += Gasagebi + "   ";
}
```

გასაღებისა და მნიშვნელობის ძებნა. ContainsKey() ფუნქცია გამოიყენება დახარისხებულ სიაში მითითებული გასაღების მოსაძებნად. პოვნის შემთხვევაში გაიცემა true მნიშვნელობა, წინააღმდეგ შემთხვევაში კი - false. მისი სინტაქსია:

***sortedList*.ContainsKey(*გასაღები*)**

ContainsValue() ფუნქცია გამოიყენება დახარისხებულ სიაში მითითებული მნიშვნელობის მოსაძებნად. პოვნის შემთხვევაში გაიცემა true, წინააღმდეგ შემთხვევაში - false. მისი სინტაქსია:

***sortedList*.ContainsValue(*მნიშვნელობა*)**

მოცემული პროგრამით ხდება ამ ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      ContainsKey() და ContainsValue()
//      ფუნქციებთან მუშაობის დემონსტრირება
SortedList^ DaxarisxebuliSia = gcnew SortedList();
//
DaxarisxebuliSia->Add(L"კა", L"კახეთი");
DaxarisxebuliSia->Add(L"იმ", L"იმერეთი");
DaxarisxebuliSia->Add(L"თუ", L"თუშეთი");
DaxarisxebuliSia->Add(L"ხე", L"ხევსურეთი");
for each ( String^ Gasagebi in DaxarisxebuliSia->Keys )
        label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in DaxarisxebuliSia->Values )
        label1->Text += Mnishvneloba->ToString() + "\n";
//
if ( DaxarisxebuliSia->ContainsKey(L"კა") )
label2->Text = L"დახარისხებული სია შეიცავს მითითებულ გასაღებს";
else    label2->Text = L"დახარისხებული სია არ შეიცავს მითითებულ გასაღებს";
//
if ( DaxarisxebuliSia->ContainsValue(L"ხევსურეთი") )
label2->Text = L"დახარისხებული სია შეიცავს მითითებულ მნიშვნელობას";
else    label2->Text = L"დახარისხებული სია არ შეიცავს მითითებულ მნიშვნელობას";
}
```

გასაღებებისა და მნიშვნელობების გადაწერა ერთგანზომილებიან მასივში. CopyTo() ფუნქცია გამოიყენება დახარისხებული სიიდან გასაღებებისა და მნიშვნელობების გადასაწერად ერთგანზომილებიან მასივში. მისი სინტაქსია:

***sortedList*.Keys.CopyTo(*მასივი, ინდექსი*)**

***sortedList*.Values.CopyTo(*მასივი, ინდექსი*)**

Keys თვისების CopyTo ფუნქცია დაწყებული ინდექსი პოზიციიდან მითითებულ მასივში გადაწერს დახარისხებული სიის გასაღებებს. Values თვისების CopyTo ფუნქცია დაწყებული ინდექსი პოზიციიდან მითითებულ მასივში გადაწერს დახარისხებული სიის მნიშვნელობებს.

მოცემული პროგრამით ხდება ამის დემონსტრირება.

```
{
//      CopyTo()ფუნქციასთან მუშაობის დემონსტრირება
SortedList^ DaxarisxebuliSia = gcnew SortedList();
//
DaxarisxebuliSia->Add(L"კა", L"კახეთი");
DaxarisxebuliSia->Add(L"იმ", L"იმერეთი");
DaxarisxebuliSia->Add(L"თუ", L"თუშეთი");
DaxarisxebuliSia->Add(L"ხე", L"ხევსურეთი");
array<String^>^ Gasagebebi = gcnew array<String^> (DaxarisxebuliSia->Count);
array<String^>^ Mnishvnelobebi = gcnew array<String^> (DaxarisxebuliSia->Count);
```

```
for each ( String^ Gasagebi in DaxarisxebuliSia->Keys )
        label1->Text += Gasagebi->ToString() + "\n";
for each ( String^ Mnishvneloba in DaxarisxebuliSia->Values )
        label1->Text += Mnishvneloba->ToString() + "\n";
//
DaxarisxebuliSia->Keys->CopyTo(Gasagebebi, 0);
for each ( String^ Gasagebi in Gasagebebi )
        label2->Text += Gasagebi + "\n";
//
DaxarisxebuliSia->Values->CopyTo(Mnishvnelobebi, 0);
for each ( String^ Mnishvneloba in Mnishvnelobebi )
        label3->Text += Mnishvneloba + "\n";
}
```

რიგი

რიგი არის მეხსიერების უბანი, რომელიც მუშაობს პრინციპით "პირველი მოვიდა - პირველი წავიდა" (FIFO - First In, First Out). რიგი განსაზღვრულია Queue კლასში. ამ კლასის თვისებები და ფუნქციები მოცემულია 13.7 და 13.8 ცხრილებში.

მოცემული პროგრამით ხდება **Enqueue(), Dequeu()** და **Peek()** ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      Enqueue(), Dequeu() და Peek() ფუნქციებთან მუშაობის დემონსტრირება
//      რიგის შექმნა
Queue^ Rigi = gcnew Queue();
//      რიგში ელემენტების დამატება
Rigi->Enqueue(L"ანა");
Rigi->Enqueue(L"საბა");
Rigi->Enqueue(L"ლიკა");
Rigi->Enqueue(L"რომანი");
//      რიგის ელემენტების გამოტანა
for each ( String^ striqoni in Rigi )
     label1->Text += striqoni + "\n";
//      რიგში ელემენტების რაოდენობის მიღება
int ElementebisRaodenoba = Rigi->Count;
for ( int count = 0; count < ElementebisRaodenoba; count++ )
  {
    label2->Text += "Peek()   = " + Rigi->Peek()    + "\n";
    label2->Text += "Dequeu() = " + Rigi->Dequeue() + "\n";
  }
}
```

ცხრილი 13.7. Queue კლასის თვისებები

თვისება	ტიპი	აღწერა
Count	int	შეიცავს რიგში რეალურად არსებული ელემენტების რაოდენობას
IsReadOnly	bool	ამოწმებს, არის თუ არა რიგი მხოლოდ წაკითხვადი

ცხრილი 13.8. Queue კლასის ფუნქციები

თვისება	დაბრუნებული ტიპი	აღწერა
Clear()	void	რიგიდან ყველა ელემენტს შლის
Clone()	Object^	ქმნის რიგის ასლს
Contains(Object^ *ობიექტი*)	bool	ამოწმებს, არის თუ არა რიგში მითითებული ელემენტი
CopyTo(Array^ *მასივი*, int *ინდექსი*)	void	რიგის ელემენტებს გადაწერს ერთგანზომილებიან მასივში
Dequeu()	Object^	გასცემს რიგის პირველი ელემენტის მნიშვნელობას და მას რიგიდან შლის
Enqueue(Object^ *ობიექტი*)	void	რიგს ბოლოში უმატებს ელემენტს
Equals(Object^ *ობიექტი*)	bool	ამოწმებს, ტოლია თუ არა ორი რიგი
GetEnumerator()	IEnumerator^	გასცემს ჩამომთვლელს რიგის ელემენტების გასწვრივ იტერაციისათვის
GetType()	Type^	გასცემს რიგის მიმდინარე ელემენტის ტიპს
Peek(Object^ *ობიექტი*)	Object^	გასცემს რიგის პირველი ელემენტის მნიშვნელობას, მაგრამ არ შლის მას რიგიდან
ToArray()	array<Object^>^	რიგის ელემენტებს გადაწერს მასივში
ToString()	String^	გასცემს რიგის მიმდინარე ელემენტის შესაბამის სტრიქონს

სტეკი

სტეკი არის მეხსიერების უბანი, რომელიც მუშაობს პრინციპით "უკანასკნელი მოვიდა - პირველი გავიდა" (LIFO - Last In, First Out). სტეკი განსაზღვრულია Stack კლასში. ამ კლასის თვისებები და ფუნქციები მოცემულია 13.9 და 13.10 ცხრილებში.

ცხრილი 13.9. Stack კლასის თვისებები

თვისება	ტიპი	აღწერა
Count	int	შეიცავს სტეკში რეალურად არსებული ელემენტების რაოდენობას
IsReadOnly	bool	ამოწმებს, არის თუ არა სტეკი მხოლოდ წაკითხვადი

ცხრილი 13.10. Stack კლასის ფუნქციები

თვისება	დაბრუნებული ტიპი	აღწერა
Clear()	void	სტეკიდან ყველა ელემენტს შლის
Clone()	Object^	ქმნის სტეკის ასლს
Contains(Object^ *ობიექტი*)	bool	ამოწმებს, არის თუ არა სტეკში მითითებული ელემენ-ტი
CopyTo(Array^ *მასივი*, int *ინდექსი*)	void	სტეკის ელემენტებს გადაწერს ერთგანზომილებიან მასივში
Equals(Object^ *ობიექტი*)	bool	ამოწმებს, ტოლია თუ არა ორი სტეკი
GetEnumerator()	IEnumerator^	გასცემს ჩამომთვლელს სტეკის ელემენტების გასწვრივ იტერაციისათვის
GetType()	Type^	გასცემს სტეკის მიმდინარე ელემენტის ტიპს
Peek()	Object^	გასცემს სტეკის უკანასკნელ (მწვერვალზე მყოფ) ობიექტს, მაგრამ არ შლის მას სტეკიდან
Pop()	Object^	გასცემს სტეკის უკანასკნელ (მწვერვალზე მყოფ) ობიექტს და შლის მას სტეკიდან
Push(Object^ *ობიექტი*)	void	ამატებს ელემენტს სტეკის მწვერვალზე
ToArray()	array<Object^>^	სტეკის ელემენტებს გადაწერს მასივში
ToString()	String^	გასცემს სტეკის მიმდინარე ელემენტის შესაბამის სტრიქონს

მოცემული პროგრამით ხდება **Push(), Pop()** და **Peek()** ფუნქციებთან მუშაობის დემონსტრირება.

```
{
//      Push(), Pop() და Peek() ფუნქციებთან მუშაობის დემონსტრირება
//      სტეკის შექმნა
Stack^ Steki = gcnew Stack();
//      სტეკში ელემენტების დამატება
Steki->Push(L"ანა");
Steki->Push(L"საბა");
Steki->Push(L"ლიკა");
Steki->Push(L"რომანი");
//      სტეკის ელემენტების გამოტანა
for each ( String^ striqoni in Steki )
      label1->Text += striqoni + "\n";
//      სტეკის ელემენტების რაოდენობის მიღება
int ElementebisRaodenoba = Steki->Count;
label1->Text += L"ელემენტების რაოდენობა - " +  ElementebisRaodenoba.ToString() + "\n";
for ( int count = 0; count < ElementebisRaodenoba; count++ )
{
   label2->Text += "Peek() = " + Steki->Peek() + "\n";
```

```
    label2->Text += "Pop()  = " + Steki->Pop()  + "\n";
}
}
```

ბიტების მასივი

ბიტების მასივი არის ბულის ცვლადების (true და false) მასივი. მასში true მნიშვნელობას შეესაბამება 1, false მნიშვნელობას კი - 0. ბიტების მასივი განსაზღვრულია BitArray კლასში. ამ კლასში წარმოდგენილ ფუნქციებს შორის აღსანიშნავია და, ან, არა და გამომრიცხავი ან ფუნქციები. BitArray კლასის თვისებები და ფუნქციები მოცემულია 13.11 და.13.12 ცხრილებში.

განვიხილოთ ზოგიერთი ფუნქცია.

Not() ფუნქცია გამოიყენება ბიტების მასივის ელემენტების ინვერტირებისათვის (იხ. ბიტობრივი ოპერატორები, თავი 4). მისი სინტაქსია:

bitArray1.Not()

bitArray1 არის BitArray ტიპის ობიექტი.

Or() ფუნქცია გამოიყენება „ან“ ოპერაციის შესასრულებლად მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის. მისი სინტაქსია:

bitArray1.Or(bitArray2)

And() ფუნქცია გამოიყენება „და“ ოპერაციის შესასრულებლად მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის. მისი სინტაქსია:

bitArray1.And(bitArray2)

Xor() ფუნქცია გამოიყენება „გამომრიცხავი ან“ ოპერაციის შესასრულებლად მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის. მისი სინტაქსია:

bitArray1.Xor(bitArray2)

ცხრილი 13.11. BitArray კლასის თვისებები

თვისება	ტიპი	აღწერა
Count	int	შეიცავს ბიტების მასივში რეალურად არსებული ელემენტების რაოდენობას
IsReadOnly	bool	ამოწმებს, არის თუ არა ბიტების მასივი მხოლოდ წაკითხვადი
Item	bool	განსაზღვრავს ელემენტის მნიშვნელობას მითითებული ინდექსის მიხედვით. ეს თვისება არის BiTArray კლასის ინდექსატორი
Length	int	განსაზღვრავს ელემენტების რაოდენობას, რომლებიც შეიძლება შევინახოთ ბიტების მასივში

ცხრილი 13.12. BitArray კლასის ფუნქციები

ფუნქცია	დაბრუნებული ტიპი	აღწერა
And(BitArray^ *სიდიდე*)	BitArray^	ასრულებს „და“ ოპერაციას მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის
Clone()	Object^	ქმნის ბიტების მასივის ასლს
CopyTo(Array^ *მასივი*, int *ინდექსი*)	void	ბიტების მასივის ელემენტებს ერთგანზომილებიან მასივში გადაწერს
Equals(Object^ *ობიექტი*)	bool	ამოწმებს ტოლია თუ არა ბიტების ორი მასივი
Get(int *ინდექსი*)	bool	გასცემს ბიტების მასივის მითითებული პოზიციის ბიტის მნიშვნელობას
GetEnumerator()	IEnumerator^	გასცემს ჩამომთვლელს ბიტების მასივისათვის
GetType()	Type^	გასცემს მიმდინარე ობიექტის ტიპს
Not()	BitArray^	ახდენს ბიტების მასივის ელემენტების ინვერსიას
Or(BitArray^ *სიდიდე*)	BitArray^	ასრულებს „ან“ ოპერაციას მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის
Set(int *ინდექსი*, bool *სიდიდე*)	void	ბიტების მასივის მითითებული პოზიციის ბიტს ანიჭებს მითითებულ მნიშვნელობას
SetAll(bool *სიდიდე*)	void	მითითებულ მნიშვნელობას ანიჭებს ბიტების მასივის ყველა ელემენტს
ToString()	String^	გასცემს მიმდინარე ელემენტის შესაბამის სტრიქონს
Xor(BitArray^ *სიდიდე*)	BitArray^	ასრულებს „გამომრიცხავ ან“ ოპერაციას მიმდინარე ბიტების მასივის ელემენტებსა და პარამეტრად გადაცემული ბიტების მასივის ელემენტებს შორის

მოცემული პროგრამით ხდება ამ ფუნქციების მუშაობის დემონსტრირება.

```
{
//      Not(),Or(), And() და Xor() ფუნქციებთან მუშაობის დემონსტრირება
label1->Text = L"BitebisMasivi";
label2->Text = L"BitebisMasivi1";
label7->Text = L"NOT";
label4->Text = L"OR";
label5->Text = L"AND";
label6->Text = L"XOR";
BitArray^ BitebisMasivi = gcnew BitArray(4);
BitebisMasivi[0] = true;
BitebisMasivi[1] = false;
BitebisMasivi[2] = true;
BitebisMasivi[3] = false;
//      ელემენტების გამოტანა
label1->Text += "\n";
label2->Text += "\n";
```

```
label7->Text += "\n";
label4->Text += "\n";
label5->Text += "\n";
label6->Text += "\n";
for ( int indexi = 0; indexi < BitebisMasivi->Count; indexi++ )
label1->Text += L"BitebisMasivi[" + indexi.ToString() + L"] = " +  BitebisMasivi[indexi].ToString() + "\n";
//
BitArray^ BitebisMasivi1 = gcnew BitArray(BitebisMasivi);
BitebisMasivi1[0] = true;
BitebisMasivi1[1] = true;
BitebisMasivi1[2] = false;
BitebisMasivi1[3] = false;
//          ელემენტების გამოტანა
for ( int indexi = 0; indexi < BitebisMasivi1->Count; indexi++ )
   label2->Text += "BitebisMasivi1[" + indexi.ToString() + "] = " +
              BitebisMasivi1[indexi].ToString() + "\n";
//          Not() ფუნქცია ახდენს BitebisMasivi მასივის ელემენტების ინვერსიას
BitebisMasivi->Not();               //          BitebisMasivi = { false, true, false, true } ელემენტების გამოტანა
for ( int indexi = 0; indexi < BitebisMasivi->Count; indexi++ )
   label7->Text += "BitebisMasivi[" + indexi.ToString() + "] = " +  BitebisMasivi[indexi].ToString() + "\n";
//          Or() ფუნქცია BitebisMasivi და BitebisMasivi1 მასივის ელემენტებს შორის
//          ასრულებს ან ოპერაციას
BitebisMasivi->Or(BitebisMasivi1);     // BitebisMasivi = { true,true,false,true } ელემენტების გამოტანა
for ( int indexi = 0; indexi < BitebisMasivi->Count; indexi++ )
   label4->Text += "BitebisMasivi[" + indexi.ToString() + "] = " +  BitebisMasivi[indexi].ToString() + "\n";
//          And() ფუნქცია BitebisMasivi და BitebisMasivi1 მასივის ელემენტებს შორის
//          ასრულებს და ოპერაციას
BitebisMasivi->And(BitebisMasivi1);              // BitebisMasivi = { true,true,false,false } ელემენტების
                                                 // გამოტანა
for ( int indexi = 0; indexi < BitebisMasivi->Count; indexi++ )
   label5->Text += "BitebisMasivi[" + indexi.ToString() + "] = " +  BitebisMasivi[indexi].ToString() + "\n";
//          Xor() ფუნქცია BitebisMasivi და BitebisMasivi1 მასივის ელემენტებს შორის ასრულებს
//          გამომრიცხავ ან ოპერაციას
BitebisMasivi->Xor(BitebisMasivi1);              // BitebisMasivi = {false,false,false,false} ელემენტების
                                                 //  გამოტანა
for ( int indexi = 0; indexi < BitebisMasivi->Count; indexi++ )
  label6->Text += "BitebisMasivi[" + indexi.ToString() + "] = " +  BitebisMasivi[indexi].ToString() + "\n";
}
```

თავი 15. შესრულების ნაკადი

შესრულების ნაკადის განსაზღვრა

პროცესი არის შესრულების სტადიაში მყოფი პროგრამა (პროგრამა-დანართი). მაგალითად, თუ ჩვენ გავუშვებთ Word ტექსტურ რედაქტორს, მაშინ შეიქმნება მისი შესაბამისი პროცესი. ოპერაციული სისტემები პროცესებს იყენებენ პროგრამა-დანართების, როგორიცაა Word, Excel და ა.შ., განცალკევებისათვის. თითოეული პროცესი ქმნის ერთ ან მეტ ნაკადს პროგრამის კოდის ნაწილების შესრულების მართვის მიზნით. ***ნაკადი*** არის ძირითადი ერთეული, რომლისთვისაც ოპერაციული სისტემა პროცესორის დროს გამოყოფს. პროცესები და ნაკადები დაწვრილებითაა განხილული ლიტერატურაში [9] და [10].

.NET Framework სისტემა პროცესს ყოფს ადვილად სამართავ ქვეპროცესებად, რომლებსაც ***პროგრამა-დანართების დომენები*** ეწოდება. ეს არის იზოლირებული გარემო, რომელშიც პროგრამა-დანართი სრულდება. პროგრამა-დანართის დომენები, რომლებიც წარმოდგენილი არიან AppDomain ობიექტებით, უზრუნველყოფენ იზოლირებისა და უსაფრთხოების საზღვრებს შესრულების პროცესში მყოფი კოდისთვის. ერთი პროცესის შიგნით ერთი ან მეტი ნაკადი შეიძლება შესრულდეს ერთ ან მეტ პროგრამა-დანართის დომენში.

ნაკადებთან სამუშაო კლასები განსაზღვრულია System::Threading სახელების სივრცეში. ნაკადი ორგვარია: ***შესრულების ნაკადი*** და ***მონაცემების ნაკადი***. მონაცემების ნაკადი ესაა ნაკადი, რომლის გადაადგილება ხდება ერთი მოწყობილობიდან მეორეზე (იგი მე-9 თავში განვიხილეთ). ამ თავში დაწვრილებით განვიხილავთ შესრულების ნაკადს.

ცხრილი 14.1. Thread კლასის თვისებები

თვისება	ტიპი	აღწერა
ApartmentState	ApartmentState	განსაზღვრავს ნაკადი ერთნაკადურ გარემოში სრულდება თუ მრავალნაკადურში
CurrentCulture	CultureInfo^	ნაკადისათვის აყენებს ან გასცემს რეგიონალურ პარამეტრებს
CurrentThread	Thread^	გასცემს ნაკადს, რომელიც მოცემულ მომენტში სრულდება
IsAlive	bool	გასცემს true მნიშვნელობას თუ ნაკადი იქნა გაშვებული და არ დამთავრებულა
IsBackground	bool	ნაკადისთვის აყენებს ან გასცემს ფონური შესრულების ატრიბუტს
IsThreadPoolThread	bool	გასცემს true მნიშვნელობას თუ ნაკადი იმყოფება ნაკადების სისტემურ პულში
Name	String^	აყენებს ან გასცემს ნაკადის სახელს. თუ სახელი მითითებული არ არის, მაშინ ის არის null-ის ტოლი
Priority	ThreadPriority	გასცემს ან აყენებს ნაკადის პრიორიტეტს
ThreadState	ThreadState	გასცემს ნაკადის მდგომარეობას

ცხრილი 14.2. Thread კლასის ფუნქციები

ფუნქცია	აღწერა
Abort()	წყვეტს ნაკადის შესრულებას
LocalDataStoreSlot^ AllocateDataSlot() (სტატიკურია)	ყველა ნაკადისთვის გამოყოფს მონაცემების არასახელდებულ უბანს
LocalDataStoreSlot^ AllocateNamedDataSlot(String^ სახელი) (სტატიკურია)	ყველა ნაკადისთვის გამოყოფს მონაცემების სახელდებულ უბანს
bool Equals(Object^ *ობიექტი*)	ამოწმებს, ტოლია თუ არა ორი ობიექტი
void FreeNamedDataSlot(String^ *სახელი*) (სტატიკურია)	ათავისუფლებს ადრე გამოყოფილ მონაცემების სახელდებულ უბანს
Object^ GetData(LocalDataStoreSlot^ სლოტი) (სტატიკურია)	გასცემს მონაცემების უბნის მნიშვნელობას
AppDomain^ GetDomain() (სტატიკურია)	გასცემს დომენს, რომელშიც ნაკადი სრულდება
int GetDomainID() (სტატიკურია)	გასცემს იმ დომენის იდენტიფიკატორს, რომელშიც ნაკადი სრულდება
LocalDataStoreSlot^ GetNamedDataSlot(String^ სახელი) (სტატიკურია)	გასცემს მონაცემების სახელდებული უბნის მნიშვნელობას
Type^ GetType()	გასცემს მიმდინარე ობიექტის ტიპს
void Interrupt()	წყვეტს ნაკადის შესრულებას
Join()	ბლოკავს ნაკადის შესრულებას სხვა ნაკადის დამთავრებამდე
void ResetAbort() (სტატიკურია)	აუქმებს მიმდინარე ნაკადის შეწყვეტის მოთხოვნას
void Resume()	აგრძელებს შეჩერებული ნაკადის შესრულებას
void SetData(LocalDataStoreSlot^ *სლოტი*, Object^ *სიდიდე*) (სტატიკურია)	მონაცემების უბანს მნიშვნელობას ანიჭებს
void Sleep(int *მილიწამები*)	აჩერებს ნაკადის შესრულებას მითითებული დროის განმავლობაში
void SpinWait(int *რაოდენობა*) (სტატიკურია)	ნაკადს აიძულებს გამოტოვოს ციკლების მითითებული რაოდენობა
void Start()	იწყებს ნაკადის შესრულებას
void Suspend()	დროებით აჩერებს ნაკადის შესრულებას
String^ ToString()	გასცემს მიმდინარე ნაკადის შესაბამის სტრიქონს

ხშირად კომპიუტერში რამდენიმე პროგრამა ერთდროულად სრულდება. მაგალითად, ჩვენ შეგვიძლია ერთმანეთის მიყოლებით შესასრულებლად გავუშვათ Excel, Paint და Internet Explorer პროგრამები. თითოეული მათგანი წარმოდგენილი იქნება შესაბამისი პროცესით. პროცესორი ამ პროცესებს შორის ისე სწრაფად ახდენს გადართვას, რომ ჩვენ გვეჩვენება თითქოს

სამივე პროგრამა ერთდროულად სრულდება. სინამდვილეში, პროცესორი ასრულებს პირველი პროცესის კოდის მცირე ნაწილს, შემდეგ მეორე პროცესის კოდის მცირე ნაწილს და ა.შ. თითოეული პროცესი წარმოქმნის ძირითად ნაკადს, რომელშიც ხდება შესაბამისი პროგრამის შესრულება.

ახლა დავუშვათ, რომ Excel პროგრამასთან მუშაობისას ჩვენ შევასრულეთ პრინტერზე ბეჭდვის ბრძანება. ამ შემთხვევაში, პროცესი წარმოქმნის მეორე ნაკადს, რომელშიც შესრულდება პრინტერზე მონაცემების ბეჭდვა. ამავე დროს ძირითად ნაკადში ჩვენ შეგვიძლია გავაგრძელოთ ცხრილის რედაქტირება. პროცესორი მოახდენს ამ ორ ნაკადს შორის გადართვას.

Thread კლასში განსაზღვრული თვისებები და ფუნქციები მოცემულია 14.1 და 14.2 ცხრილებში.

შესრულების ნაკადის შექმნა

ნაკადის შესაქმნელად უნდა შევქმნათ Thread კლასის ობიექტი. ამ კლასის კონსტრუქტორს უნდა გადავცეთ დელეგატი. ეს დელეგატი მიმართავს იმ ფუნქციას, რომელიც უნდა შესრულდეს ნაკადის ქვეშ. პროგრამის საწყისი კოდი, რომელშიც სრულდება ნაკადებთან მუშაობა, უნდა იწყებოდეს დირექტივით:

```
using namespace System::Threading;
```

ამ განყოფილებაში შევადგენთ კონსოლურ პროგრამა-დანართებს, რადგან ასეთ პროგრამებთან მუშაობისას ნაკადების შესრულების შედეგები უფრო მკაფიოდ ჩანს. კონსოლური პროგრამის შესაქმნელად 2.2 ნახ.-ზე ნაჩვენები ფანჯრის Templates ზონაში უნდა მოვნიშნოთ CLR Console Application ელემენტი, Name: ველში შევიტანოთ პროგრამის სახელი, კატალოგი ავირჩიოთ Browse კლავიშის საშუალებით და დავაჭიროთ OK კლავიშს. გახსნილ ფანჯარაში შეგვაქვს ჩვენი პროგრამის კოდი ისე, როგორც ეს ქვემოთაა ნაჩვენები.

```
//      ორ ნაკადთან მუშაობის დემონსტრირება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi {
//      ნაკადში შესასრულებელი ფუნქციის გამოძახება
public : static void Datvla_Luwi() {
for ( int index2 = 2; index2 <= 50; index2 += 2 )
Console::Write(index2.ToString() + "  ");
Thread::Sleep(5000);            //      ეკრანზე შედეგების დაყოვნება 5 წამის განმავლობაში
}
};
//
int main(array<System::String ^> ^args) {
//      Nakadi1 ნაკადის შექმნა, რომელშიც შესრულდება Datvla_Luwi() ფუნქცია
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::Datvla_Luwi));
//      Nakadi1 ნაკადში შესრულებას იწყებს Datvla_Luwi() ფუნქცია
Nakadi1->Start();
//      პირველ, ძირითად ნაკადში შესრულებას იწყებს Datvla_Luwi() ფუნქცია
ChemiKlasi::Datvla_Luwi();
```

}

პროგრამით იქმნება Nakadi1 ობიექტი-ნაკადი. კონსტრუქტორს გადაეცემა Datvla_Luwi დელეგატი, რომელიც არის Datvla_Luwi() ფუნქციაზე მიმთითებელი. ამ ფუნქციის შესრულების დასაწყებად გამოიძახება Nakadi1 ობიექტის Start() ფუნქცია. ამის შემდეგ, მომდევნო სტრიქონში ვიძახებთ Datvla_Luwi() ფუნქციას, რომელიც პირველ ძირითად ნაკადში შესრულდება. შედეგად, ორ სხვადასხვა ნაკადში ერთდროულად სრულდება Datvla_Luwi() ფუნქცია. პროგრამის მუშაობის ერთ-ერთი შესაძლო შედეგი:

```
 2   4   6   8   10  12  14  16  18  20  22  24  26  28  30  32  34  2   4   6   8
 10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40  42  44  46  48  50
 36  38  40  42  44  46  48  50
```

აქ ორი ნაკადის მუშაობის შედეგები ერთმანეთშია არეული. ეს იმიტომ მოხდა, რომ ორივე ნაკადს შედეგები ერთ მოწყობილობაზე გამოაქვს. ჩვენ შემთხვევაში ეს მოწყობილობაა - მონიტორი. როგორც ვხედავთ, პირველმა ნაკადმა, რომელშიც Datvla_Luwi() ფუნქცია სრულდება, მონაცემები გამოიტანა 2-დან 34-ის ჩათვლით. შემდეგ პროცესორი გადაერთო Nakadi1-ის შესრულებაზე. ამ ნაკადში მუშაობს Datvla_Luwi() ფუნქცია. ამ ნაკადმა მონაცემები თავიდან ბოლომდე გამოიტანა ეკრანზე. შემდეგ პროცესორმა ისევ გააგრძელა პირველი ნაკადის შესრულება, რომელმაც დაამთავრა ეკრანზე მონაცემების გამოტანა.

აქვე შევნიშნოთ, რომ ამ პროგრამის ყოველი გაშვებისას ეკრანზე შეიძლება სხვადასხვა შედეგი მივიღოთ. საქმე ისაა, რომ წინასწარ უცნობია თუ როდის მოახდენს პროცესორი ერთი ნაკადიდან მეორეზე გადართვას.

რაც შეეხება Thread::Sleep(5000) ფუნქციას, ის საშუალებას გვაძლევს შედეგები ეკრანზე დავაყოვნოთ 5 წამის განმავლობაში (1000 შეესაბამება 1 წამს).

იმისათვის, რომ პროგრამის შესრულების შედეგები უფრო თვალსაჩინო იყოს, ქვემოთ მოცემული პროგრამის ChemiKlasi კლასში გამოვაცხადოთ ორი ფუნქცია: Datvla_Kenti() და Datvla_Luwi(), რომლებიც შესაბამისად, კენტ და ლუწ რიცხვებს გასცემენ.

```
//      ორი სხვადასხვა ფუნქციის ორ სხვადასხვა ნაკადში
//      შესრულების დემონსტრირება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi {
public : static void Datvla_Kenti() {
for (int index1 = 1; index1 <= 50; index1 += 2)
Console::Write(index1.ToString() + "  ");
Thread::Sleep(5000);
}
static void Datvla_Luwi() {
for ( int index2 = 2; index2 <= 50; index2 += 2 )
        Console::Write(index2.ToString() + "  ");
Thread::Sleep(5000);
}
};
//
```

```
int main(array<System::String ^> ^args) {
//
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::Datvla_Luwi));
//        Datvla_Luwi()ფუნქცია შესრულებას იწყებს Nakadi1 ნაკადში
Nakadi1->Start();
//        Datvla_Kenti() ფუნქცია შესრულებას იწყებს ძირითად ნაკადში
ChemiKlasi::Datvla_Kenti();
}
```

ძირითადი პროგრამით იქმნება ობიექტი-ნაკადი Nakadi1. კონსტრუქტორს გადაეცემა დელეგატი - Datvla_Luwi, რომელიც წარმოადგენს Datvla_Luwi() ფუნქციაზე მიმთითებელს. შემდეგ, Datvla_Luwi ფუნქციის შესრულების დასაწყებად ვიძახებთ Nakadi1 ობიექტის Start() ფუნქციას. ამის შემდეგ, მომდევნო სტრიქონში ვიძახებთ Datvla_Kenti() ფუნქციას, რომელიც პირველ, ძირითად ნაკადში შესრულდება. შედეგად, ორ სხვადასხვა ნაკადში ერთდროულად სრულდება Datvla_Kenti() და Datvla_Luwi() ფუნქციები. პროგრამის მუშაობის ერთ-ერთი შესაძლო შედეგი:

```
 1   3   5   7   9   11  13  15  17  19  21  23  25  27  29  31  2   4   6   8   10
 12  14  16  18  20  22  24  26  28  30  32  34  36  38  40  42  44  46  48  50  33
 35  37  39  41  43  45  47  49
```

შესრულების ნაკადის პრიორიტეტები

შექმნისას თითოეულ ნაკადს ენიჭება Normal პრიორიტეტი. სულ არსებობს პრიორიტეტების 5 დონე:

Lowest - უდაბლესი
BelowNormal - ნორმალურზე დაბალი
Normal - ნორმალური
AboveNormal - ნორმალურზე მაღალი
Highest - უმაღლესი

მოცემული პროგრამით ხდება ნაკადებისათვის პრიორიტეტების მინიჭების დემონსტრირება.

```
//        ნაკადებისათვის პრიორიტეტების მინიჭების დემონსტრირება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi {
public : static void Datvla_Kenti() {
for ( int index1 = 1; index1 <= 50; index1 += 2 )
        Console::Write(index1.ToString() + "  ");
Thread::Sleep(5000);
}
static void Datvla_Luwi() {
for ( int index2 = 2; index2 <= 50; index2 += 2 )
        Console::Write(index2.ToString() + "  ");
```

```
Thread::Sleep(5000);
}
};
//
int main(array<System::String ^> ^args) {
 //        Nakadi1 ნაკადს შექმნისას ენიჭება Normal პრიორიტეტი
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::Datvla_Kenti));
//        მიმდინარე ნაკადს ენიჭება Lowest პრიორიტეტი
Thread::CurrentThread->Priority = ThreadPriority::Lowest;
//        Nakadi1 ნაკადს შექმნისას ენიჭება Highest პრიორიტეტი
Nakadi1->Priority = ThreadPriority::Highest;
Nakadi1->Start();
ChemiKlasi::Datvla_Luwi();
}
```

პროგრამაში Nakadi1 ნაკადს უმაღლესი პრიორიტეტი ენიჭება, ხოლო ძირითად ნაკადს, რომელშიც Datvla_Luwi() ფუნქცია სრულდება - უდაბლესი. აქ Thread კლასის CurrentThread სტატიკური თვისება გასცემს მიმდინარე ნაკადს ანუ იმ ნაკადს, რომელიც მოცემულ მომენტში სრულდება, Thread ობიექტის CurrentThread თვისება კი - მიმდინარე ნაკადზე მიმთითებელს. პროგრამის შესრულების შედეგები:

```
 1   3   5   7   9   11  13  15  17  19  21  23  25  27  29  31  33  35  37  39  41
 43  45  47  49  2   4   6   8   10  12  14  16  18  20  22  24  26  28  30  32  34
 36  38  40  42  44  46  48  50
```

როგორც პროგრამის შესრულების შედეგებიდან ჩანს, ჯერ მუშაობას ამთავრებს პრიორიტეტული ნაკადი, შემდეგ კი - დაბალი პრიორიტეტის მქონე ნაკადი. თუმცა არ არსებობს იმის გარანტია, რომ მაღალპრიორიტეტული ნაკადი უფრო ადრე დაამთავრებს შესრულებას, ვიდრე დაბალპრიორიტეტული. საქმე ის არის, რომ, ჯერ ერთი, ყველა ოპერაციული სისტემა რესურსებს ერთნაირად არ ანაწილებს. მეორეც, პრიორიტეტები ხშირად განსაზღვრავს პროცესორული დროის ხანგრძლივობას და არა პროცესების შესრულების რიგითობას. შედეგად, მაღალპრიორიტეტულმა პროცესმა, რომელიც დიდი მოცულობის გამოთვლებს ასრულებს, მუშაობა შეიძლება უფრო გვიან დაამთავროს, ვიდრე დაბალპრიორიტეტულმა, რომელიც მცირე მოცულობის გამოთვლებს ასრულებს.

შესრულების ნაკადის მდგომარეობა

ნაკადის მიმდინარე მდგომარეობის მისაღებად გამოიყენება ThreadState თვისება. ნაკადი არსებობის მანძილზე შეიძლება რამდენიმე მდგომარეობაში იმყოფებოდეს. ეს მდგომარეობები მოცემულია 14.3 ცხრილში.

შექმნის შემდეგ ნაკადს აქვს Unstarted მდგომარეობა. Start() ფუნქციის გამოძახების შემდეგ ნაკადი Running მდგომარებაში გადადის. თუ ნაკადის IsBackground თვისებას true მნიშვნელობას მივანიჭებთ, მაშინ ის ფონურ რეჟიმში იმუშავებს. ნაკადი დროის ერთსა და იმავე მომენტში შეიძლება რამდენიმე მდგომარეობაში იმყოფებოდეს, მაგალითად, WaitSleepJoin და AbortRequested მდგომარეობებში.

ცხრილი 14.3. ნაკადის მდგომარეობები.

მდგომარეობა	მდგომარეობის აღწერა
Unstarted	ნაკადს შესრულება არ დაუწყია
Running	ნაკადი სრულდება
Background	ნაკადი ფონურ რეჟიმში სრულდება
WaitSleepJoin	ნაკადი ბლოკირებულია Wait, Sleep ან Join ფუნქციის მიერ
SuspendRequested	მოთხოვნილია ნაკადის დროებით შეჩერება
Suspended	ნაკადი დროებით შეჩერებულია
StopRequested	მოთხოვნილია ნაკადის გაჩერება
Stopped	ნაკადი გაჩერებულია
AbortRequested	მოთხოვნილია ნაკადის გაუქმება
Aborted	ნაკადი გაუქმებულია

მოცემული პროგრამით ხდება ThreadState თვისებასთან მუშაობის დემონსტრირება.

```
//      ThreadState თვისებასთან მუშაობა
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi
{
public :
static void Datvla()
{
for ( int index = 1; index <= 50; index++ )
Console::Write(index.ToString() + "  ");
Console::WriteLine();
Thread::Sleep(5000);
}
static void MdgomareobisNaxva(Thread^ Nakadi)
{
if (  Nakadi->ThreadState == ThreadState::Running )
        Console::WriteLine(L" The Thread is running");
}
};
//
int main(array<System::String ^> ^args)
{
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::Datvla));
ChemiKlasi::MdgomareobisNaxva(Nakadi1);
//      ნაკადის გაშვება
Nakadi1->Start();
ChemiKlasi::MdgomareobisNaxva(Nakadi1);
//      ამავე დროს სრულდება მეორე ნაკადი
ChemiKlasi::Datvla();
```

```
//        nakadi1 ნაკადის შეწყვეტა
Nakadi1->Abort();
ChemiKlasi::MdgomareobisNaxva(Nakadi1);
}
```

პროგრამის შესრულების შედეგის ერთ-ერთი შესაძლო ვარიანტი:

```
 The Thread is running
 1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21
 22  23  24  25  26  27  28  29  30  31  32  33  34  35  1   2   3   4   5   6   7
 8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
 29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49
 50  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
```

შესრულების ნაკადის ლოკალური მონაცემები

გადასაწყვეტი ამოცანიდან გამომდინარე ნაკადმა შეიძლება გამოიყენოს ოპერატიული მეხსიერების ერთი და იგივე ან სხვადასხვა უბნები. იმისათვის, რომ რამდენიმე ნაკადმა ერთობლივად გამოიყენოს მონაცემები, საჭიროა ამ მონაცემების გამოცხადება ნაკადებისათვის საერთო ხილვადობის უბანში. იმისათვის, რომ პროცესებმა თავიანთ ლოკალურ მონაცემებთან იმუშაონ ანუ გამოიყენონ ოპერატიული მეხსიერების სხვადასხვა უბნები, უნდა გამოვიყენოთ LocalDataStoreSlot ობიექტები.

მოცემული პროგრამით ხდება ორი პროცესის მიერ საკუთარ ლოკალურ უბნებთან მუშაობის დემონსტრირება.

```
//        ორი პროცესის მიერ საკუთარ ლოკალურ უბნებთან მუშაობა
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi
{
public :
static void MonacemebisGamotana()
{
Console::WriteLine(L"Random number = " +
Thread::GetData(Thread::GetNamedDataSlot(L"Random number - ")));
Console::WriteLine(L"Name of thread      = " +
Thread::GetData(Thread::GetNamedDataSlot(L"Name of thread     - ")));
Thread::Sleep(5000);
}
static void MonacemebisChatsera1()
{
Random^ Shemtxveviti_Ricxvi = gcnew Random();
Thread::SetData(Thread::GetNamedDataSlot(L"Random number - "),
                                    Shemtxveviti_Ricxvi->Next(100));
Thread::SetData(Thread::GetNamedDataSlot(L"Name of thread      - "),
```

```
                                    Thread::CurrentThread->Name);
MonacemebisGamotana();
}
static void MonacemebisChatsera2()
{
Random^ Shemtxveviti_Ricxvi = gcnew Random();
Thread::SetData(Thread::GetNamedDataSlot(L"Random number - "),
                                    Shemtxveviti_Ricxvi->NextDouble());
Thread::SetData(Thread::GetNamedDataSlot(L"Name of thread     - "),
                                    Thread::CurrentThread->Name);
MonacemebisGamotana();
}
};
//
int main(array<System::String ^> ^args)
{
 //        სახელდებული უბნების გამოყოფა
Thread::AllocateNamedDataSlot(L"Random number - ");
Thread::AllocateNamedDataSlot(L"Name of thread      - ");
//        მეორე ნაკადის შექმნა და გაშვება
Thread^ nakadi1 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::MonacemebisChatsera1));
nakadi1->Name = "nakadi1";
nakadi1->Start();
//        მესამე ნაკადის შექმნა და გაშვება
Thread^ nakadi2 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::MonacemebisChatsera2));
nakadi2->Name = "nakadi2";
nakadi2->Start();
// მონაცემების უბნის გასუფთავება
Thread::FreeNamedDataSlot(L"Random number - ");
Thread::FreeNamedDataSlot(L"Name of thread     - ");
}
```

ძირითად პროგრამაში ხდება მეხსიერების ორი უბნის გამოყოფა AllocateNamedDataSlot() ფუნქციის საშუალებით. ამ უბნების სახელებია: "შემთხვევითი რიცხვი - " და "ნაკადის სახელი - ". შემდეგ იქმნება ორი ნაკადი: nakadi1 და nakadi2. მათ ენიჭება შესაბამისი სახელები. nakadi1 ნაკადში სრულდება MonacemebisChatsera1() ფუნქცია, nakadi2 ნაკადში კი - MonacemebisChatsera2() ფუნქცია. თითოეული ფუნქცია "შემთხვევითი რიცხვი - " უბანში წერს შემთხვევით რიცხვს, "ნაკადის სახელი - " უბანში კი - შესაბამისი ნაკადის სახელს. შემდეგ ხდება MonacemebisGamotana() ფუნქციის გამოძახება, რომელსაც ეკრანზე გამოაქვს "შემთხვევითი რიცხვი - " და "ნაკადის სახელი - " უბნებში მოთავსებული მონაცემები. ძირითადი პროგრამის დასასრულს ხდება მეხსიერების გამოყოფილი უბნების გათავისუფლება და ოპერაციული სისტემისათვის გადაცემა. პროგრამის შესრულების შედეგები:

```
Shemtxveviti ricxvi     = 4
Nakadis saxeli          = nakadi1
Shemtxveviti ricxvi     =0,400937242154468
Nakadis saxeli          = nakadi2
```

მიუხედავად იმისა, რომ ორივე ნაკადი იყენებს ერთი და იმავე სახელის მქონე მეხსიერების უბნებს, მათ მაინც გამოეყოფა მეხსიერების სხვადასხვა ლოკალური უბანი.

შესრულების ნაკადის მართვა

ჩვეულებრივ, ჩვენ არ ვიცით თუ რომელი ნაკადი როდის შესრულდება და ვერ განვსაზღვრავთ მათი შესრულების მიმდევრობას. ამ განყოფილებაში განვიხილავთ ნაკადების მართვის ხერხებს.

Timer კლასი

Timer კლასი საშუალებას გვაძლევს ნაკადი დროის გარკვეული გამეორებადი ინტერვალების შემდეგ ანუ გარკვეული პერიოდულობით შევასრულოთ. Timer ტიპის ობიექტის შექმნისას კონსტრუქტორს ოთხი პარამეტრი უნდა გადავცეთ:

– callback. ესაა TimeCallback ტიპის დელეგატი, რომელიც მიმართავს იმ ფუნქციას, რომელსაც ტაიმერი გამოიძახებს.
– state. ესაა ობიექტი, რომელიც TimeCallback ფუნქციას გადაეცემა. ამ პარამეტრს იმიტომ გადავცემთ, რომ ტაიმერისათვის მისაწვდომი იყოს რაიმე მუდმივი მდგომარეობა, წინააღმდეგ შემთხვევაში null პარამეტრს ვუთითებთ.
– dueTime. ესაა მილიწამების რაოდენობა ტაიმერის პირველ ამუშავებამდე.
– Period. ესაა მილიწამების რაოდენობა ტაიმერის ამუშავებებს შორის.

მოცემული პროგრამით ხდება Timer კლასთან მუშაობის დემონსტრირება.

```
//      Timer კლასთან მუშაობა
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi {
public :
static void Drois_Gacema(Object^ Mdgomareoba) {
        Console::WriteLine(DateTime::Now);
}
};
//
int main(array<System::String ^> ^args) {
//      დელეგატის შექმნა, რომელსაც Timer ობიექტი გამოიძახებს
TimerCallback^ delegati = gcnew TimerCallback(ChemiKlasi::Drois_Gacema);
//      ტაიმერის შექმნა, რომელიც წამში ორჯერ იმუშავებს
//      პირველი გაშვება ერთი წამის შემდეგ შესრულდება
System::Threading::Timer^ Taimeri = gcnew System::Threading::Timer(delegati, nullptr, 1000, 2000);
//      პროგრამის შესრულების დასამთავრებლად ვაჭერთ Enter კლავიშს
Console::WriteLine("Press ENTER to exit");
int ricxvi = Console::Read();
//       რესურსების გათავისუფლება
Taimeri = nullptr;
```

```
}
```

პროგრამის შესრულების შედეგი:

```
 daaWireT ENTER klaviSs dasamTavreblad
 21.01.2009 13:35:16
 21.01.2009 13:35:17
 21.01.2009 13:35:18
 21.01.2009 13:35:19
 21.01.2009 13:35:20
```

პროგრამაში იქმნება delegati დელეგატი, რომელიც გამოიყენება Drois_Gacema() ფუნქციის გამოსაძახებლად. შემდეგ იქმნება Taimeri ობიექტი-ტაიმერი, რომელიც პირველად ამუშავდება 1 წამის შემდეგ, შემდეგ კი იმუშავებს წამში ორჯერ ანუ წამში ორჯერ მოხდება Drois_Gacema() ფუნქციის გამოძახება. პროგრამის ბოლოს Dispose() ფუნქცია ათავისუფლებს ოპერაციული სისტემის რესურსებს, რომელიც ნაკადს გამოეყო შექმნისას.

Join() ფუნქცია

Join() ფუნქცია გამოიყენება ერთი ნაკადის მისაბმელად მეორე ნაკადისათვის. შედეგად, მიბმული ნაკადი დაელოდება პირველი ნაკადის შესრულების დამთავრებას და ამის შემდეგ დაიწყებს მუშაობას. მაგალითი:

```
//      ერთი ნაკადის მეორისათვის მიბმის დემონსტრირება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

ref class ChemiKlasi
{
public :
static void Datvla_Kenti() {
for ( int indexi = 1; indexi <= 10; indexi += 2 )
Console::Write(indexi.ToString() + "  ");
Thread::Sleep(3000);
}
static void Datvla_Luwi() {
for (int indexi = 2; indexi <= 10; indexi += 2)
Console::Write(indexi.ToString() + "  ");
Thread::Sleep(3000);
}
};
//
int main(array<System::String ^> ^args) {
Thread^ nakadi2 = gcnew Thread(gcnew ThreadStart(ChemiKlasi::Datvla_Kenti));
//
nakadi2->Start();
//      nakadi2-ის ბლოკირება პირველი ნაკადის დამთავრებამდე
```

```
nakadi2->Join();
ChemiKlasi::Datvla_Luwi();
}
```

პროგრამაში ხდება nakadi2 ნაკადის ბლოკირება მანამ, სანამ არ დამთავრდება Datvla_Luwi() ფუნქციის შესრულება. პროგრამის შესრულების შედეგი:

```
2        4        6        8        10        1        3        5        7        9
```

Join() ფუნქცია საშუალებას გვაძლევს, აგრეთვე, მივუთითოთ ლოდინის მაქსიმალური დრო. მაგალითად, nakadi2->Join(4000) ფუნქციის შესრულების შედეგად nakadi2 ნაკადი მეორე ნაკადის შესრულების დამთავრებას დაელოდება არაუმეტეს 4 წამისა.

Interlocked კლასი

Interlocked კლასი გამოიყენება ცვლადების უსაფრთხო ინკრემენტისა და დეკრემენტისათვის. ამ კლასის ფუნქციები ნაჩვენებია 14.4 ცხრილში.

ცხრილი 14.4. Interlocked კლასის ფუნქციები

ფუნქცია	აღწერა
int Decrement(int% *მისამართი*) (სტატიკური)	ცვლადის მნიშვნელობას ერთით ამცირებს
int Exchange(int% *მისამართი*, int *სიდიდე*) (სტატიკური)	ცვლადს მნიშვნელობას ანიჭებს
int Increment(int% *მისამართი*) (სტატიკური)	ცვლადის მნიშვნელობას ერთით ზრდის

მოცემული პროგრამით ხდება Decrement() ფუნქციის გამოყენების დემონსტრირება.

```
//      Decrement() ფუნქციის გამოყენება
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

int Saerto_Mtvleli = 10;
ref class ChemiKlasi {
public : void Datvla() {
while ( Saerto_Mtvleli > 0 ) {
        System::Threading::Interlocked::Decrement(Saerto_Mtvleli);
  Console::WriteLine(Thread::CurrentThread->Name + "   " + Saerto_Mtvleli);
        Thread::Sleep(500);
}
                Thread::Sleep(3000);
}
void Nakadebis_Shesruleba() {
ChemiKlasi^ ck = gcnew ChemiKlasi;
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ck, &ChemiKlasi::Datvla));
```

```
Nakadi1->Name = "Nakadi1";
Thread^ Nakadi2 = gcnew Thread(gcnew ThreadStart(ck, &ChemiKlasi::Datvla));
Nakadi2->Name = "Nakadi2";
Nakadi1->Start();
Nakadi2->Start();
}
};
//
int main(array<System::String ^> ^args) {
ChemiKlasi^ obieqti = gcnew ChemiKlasi();
obieqti->Nakadebis_Shesruleba();
}
```

პროგრამის შესრულების შედეგი:

```
Nakadi1     9
Nakadi2     8
Nakadi1     7
Nakadi2     6
Nakadi1     5
Nakadi2     4
Nakadi1     3
Nakadi2     2
Nakadi1     1
Nakadi2     0
```

თუ Datvla() ფუნქციაში Decrement() ფუნქციას შევცვლით Increment() ფუნქციით, მაშინ Saerto_Mtvleli ცვლადის მნიშვნელობა ერთით გაიზრდება ამ ფუნქციის ყოველი შესრულებისას. Exchange() ფუნქციის გამოყენების შემთხვევაში Datvla() ფუნქცია შემდეგ სახეს მიიღებს.

```
public : void Datvla() {
Console::WriteLine(Thread::CurrentThread->Name + "   " + Saerto_Mtvleli);
System::Threading::Interlocked::Exchange(Saerto_Mtvleli, 55);
Console::WriteLine(Thread::CurrentThread->Name + "   " + Saerto_Mtvleli);
Thread::Sleep(5000);
}
```

Monitor კლასი

Monitor კლასი გამოიყენება კოდის ნებისმიერ ობიექტთან სინქრონიზებული მიმართვის განხორციელებისათვის. ამ კლასის ფუნქციები მოცემულია 14.5 ცხრილში.

მოცემული პროგრამით ხდება Monitor კლასის გამოიყენების დემონსტრირება.

ცხრილი 14.5. Monitor კლასის ფუნქციები

ფუნქცია	აღწერა
void Enter(Object^ *ობიექტი*) (სტატიკური)	ბლოკავს მონიტორისათვის გადაცემულ ობიექტს. თუ სხვა ნაკადმა უკვე დაბლოკა ეს ობიექტი, მაშინ მიმდინარე ნაკადის შესრულება დროებით შეწყდება მანამ, სანამ სხვა ნაკადი არ გაათავისუფლებს ამ ობიექტს
void Exit(Object^ *ობიექტი*) (სტატიკური)	ახდენს ობიექტის დებლოკირებას
void Pulse(Object^ *ობიექტი*) (სტატიკური)	მომდევნო მომლოდინე ნაკადს აუწყებს იმის შესახებ, რომ მონიტორმა დროებით დაამთავრა მუშაობა ობიექტთან და ნაკადს შეუძლია მუშაობის გაგრძელება
void PulseAll(Object^ *ობიექტი*) (სტატიკური)	ყველა ნაკადს აუწყებს იმის შესახებ, რომ ობიექტი მალე გათავისუფლდება
bool TryEnter(Object^ *ობიექტი*) (სტატიკური)	ცდილობს დაბლოკოს გადაცემული ობიექტი. თუ ობიექტის ბლოკირება შესაძლებელია, მაშინ გაიცემა true, წინააღმდეგ შემთხვევაში - false
bool Wait(Object^ *ობიექტი*) (სტატიკური)	ათავისუფლებს ყველა ბლოკირებას და დროებით აჩერებს მიმდინარე ნაკადის შესრულებას მანამ, სანამ სხვა ნაკადი არ გამოიძახებს Pulse() ფუნქციას

// Monitor კლასის გამოყენება

```
#include "stdafx.h"

using namespace System;
using namespace System::Threading;

int Saerto_Mtvleli = 0;
ref class ChemiKlasi {
public : void Datvla() {
while ( Saerto_Mtvleli < 10 )
{
        System::Threading::Monitor::Enter(this);
        int Temp = Saerto_Mtvleli;
        Temp++;
        Console::WriteLine(Thread::CurrentThread->Name + "   " + Temp);
        Thread::Sleep(500);
        Saerto_Mtvleli = Temp;
        System::Threading::Monitor::Exit(this);
}
Thread::Sleep(5000);
}
void Nakadebis_Shesruleba() {
ChemiKlasi^ ck = gcnew ChemiKlasi;
Thread^ Nakadi1 = gcnew Thread(gcnew ThreadStart(ck, &ChemiKlasi::Datvla));
Nakadi1->Name = "Nakadi1";
```

```
Thread^ Nakadi2 = gcnew Thread(gcnew ThreadStart(ck, &ChemiKlasi::Datvla));
Nakadi2->Name = "Nakadi2";
Nakadi1->Start();
Nakadi2->Start();
}
};
//
int main(array<System::String ^> ^args) {
ChemiKlasi^ obieqti = gcnew ChemiKlasi();
obieqti->Nakadebis_Shesruleba();
}
```

პროგრამის შესრულების შედეგები:

```
Nakadi1     1
Nakadi2     2
Nakadi1     3
Nakadi2     4
Nakadi1     5
Nakadi2     6
Nakadi1     7
Nakadi2     8
Nakadi1     9
Nakadi2     10
Nakadi1     11
```

თავი 16. საბაზო ბიბლიოთეკის სხვა კლასები

გლობალიზაცია

.NET გარემო შეიცავს გლობალიზაციის გაფართოებულ უზრუნველყოფას. System::Globalization სახელების სივრცეში განსაზღვრულია კლასები (15.1 ცხრილი), რომლებიც საშუალებას გვაძლევს პროგრამის კოდი სწრაფად გავაწყოთ ჩვენთვის საჭირო ეროვნული კულტურის ელემენტების გამოყენებაზე. ეროვნული კულტურის ელემენტებია: ეროვნული ენა, ვალუტის სიმბოლო, დროის, თარიღისა და რიცხვის ფორმატი და ა.შ.

.NET გარემოში თითოეული ქვეყნის კულტურას შეესაბამება აღნიშვნა, რომელიც შედგება კულტურის კოდისგან, რომელსაც მოსდევს ერთი ან მეტი ქვეკულტურის კოდი. შეთანხმების თანახმად კულტურის კოდები პატარა ასოებით აღინიშნება, ხოლო ქვეკულტურის კოდები - დიდი ასოებით. მაგალითად, en-CA კოდი განსაზღვრავს ინგლისურ კულტურას კანადაში. ეს კულტურა შეიცავს ინფორმაციას კანადაში მცხოვრები ინგლისელებისათვის. pt-BR კოდი განსაზღვრავს პორტუგალიურ კულტურას ბრაზილიაში, ka-GE კოდი განსაზღვრავს ქართულ კულტურას, bs-BA-Latn კოდი განსაზღვრავს ბოსნიის კულტურას იმ მომხმარებლისათვის, რომელიც ლათინურ ანბანს იყენებს და ა.შ. აქვე შევნიშნოთ, რომ .NET გარემოში ამერიკული ინგლისური ენა და ბრიტანული ინგლისური ენა სხვადასხვა კულტურებს მიეკუთვნება, რადგან ამ ქვეყნების ვალუტის სიმბოლოები და თარიღის ფორმატები ერთმანეთისაგან განსხვავდება.

ცხრილი 15.1. System::Globalization სახელების სივრცის კლასები

კლასი	აღწერა
Calendar	აბსტრაქტული კლასია, რომელიც გვაძლევს საერთო წარმოდგენას კულტურისათვის დამახასიათებელი კალენდრის შესახებ.
CompareInfo	სტრიქონების შედარების ფუნქციების კულტურისათვის დამახასია-თებელი კოლექციაა
CultureInfo	შეიცავს ინფორმაციას კულტურის ყველა ასპექტის შესახებ
DateTimeFormatInfo	შეიცავს ინფორმაციას დროისა და თარიღის ფორმატის შესახებ
DaylightTime	შეიცავს ინფორმაციას ზაფხულისა და ზამთრის დროზე გადასვლის შესახებ
NumberFormatInfo	შეიცავს ინფორმაციას რიცხვის წარმოდგენის ფორმატის შესახებ
RegionInfo	შეიცავს ინფორმაციას განსაზღვრული რეგიონის ან ქვეყნის შესახებ
SortKey	შეიცავს ინფორმაციას, რომელიც აუცილებელია სტრიქონების დახარისხებისათვის
StringInfo	წარმოგვიდგენს ფუნქციებს სტრიქონების დაყოფისათვის ელემენ-ტებად და ამ ელემენტების გადარჩევისათვის
TextInfo	შეიცავს ინფორმაციას ტექსტის დაფორმატების შესახებ

ცხრილი 15.2. CultureInfo კლასის თვისებები

თვისება	ტიპი	აღწერა
Calendar	Calendar^	შეიცავს კალენდარს, რომელიც მოცემულ კულტურაში ავტომატურად გამოიყენება
CompareInfo	CompareInfo^	შეიცავს ინფორმაციას მოცემულ კულტურაში სტრიქონების შედარების შესახებ
CurrentCulture (სტატიკურია)	CultureInfo^	შეიცავს მოცემული ნაკადის მიმდინარე კულტურას
CurrentUICulture (სტატიკურია)	CultureInfo^	შეიცავს სამომხმარებლო ინტერფეისის მიმდინარე კულტურას
DateTimeFormat	DateTimeFormat-Info^	შეიცავს ინფორმაციას მოცემული კულტურის თარიღისა და დროის შესახებ
DisplayName	String^	შეიცავს კულტურის ასახვად სახელს
EnglishName	String^	შეიცავს კულტურის ინგლისურ დასახელებას
InstalledUICulture (სტატიკურია)	CultureInfo^	შეიცავს ოპერაციულ სისტემასთან ერთად დაინსტალირებულ კულტურას
InvariantCulture (სტატიკურია)	CultureInfo^	შეიცავს ინვარიანტულ კულტურას
IsNeutralCulture	bool	გვიჩვენებს CultureInfo ობიექტი ნეიტრალური კულტურაა თუ არა
IsReadOnly	bool	გვიჩვენებს CultureInfo ობიექტი მხოლოდ წაკითხვადია თუ არა
LCID	String^	შეიცავს კულტურის იდენტიფიკატორს CultureInfo ობიექტისთვის
Name	String^	შეიცავს კულტურის დასახელებას
NativeName	String^	შეიცავს კულტურის ეროვნულ დასახელებას
NumberFormat	NumberFormat-Info^	შეიცავს ინფორმაციას რიცხვის ფორმატის შესახებ მოცემული კულტურისათვის
OptionalCalendars	array<Calendar^>^	შეიცავს ალტერნატიული კალენდრების სიას მოცემული კულტურისათვის
Parent	CultureInfo^	შეიცავს კულტურას, რომელიც არის მშობელი მოცემული კულტურისათვის
TextInfo	TextInfo^	შეიცავს ინფორმაციას ტექსტის დაფორმატების შესახებ მოცემული კულტურისათვის
ThreeLetterISO-LanguageName	String^	შეიცავს ISO ენის კოდს მოცემული კულტურის ენისათვის
TwoLetterISO-LanguageName	String^	შეიცავს ISO ენის ორასოიან კოდს მოცემული კულტურის ენისათვის
UseUserOverride	bool	მიუთითებს CultureInfo ობიექტი იყენებს თუ არა მომხმარებლის მიერ არჩეულ გაწყობებს

ცხრილი 15.3. CultureInfo კლასის ფუნქციები

ფუნქცია	აღწერა
void ClearCachedData()	აახლებს ქეშირებულ ინფორმაციას კულტურის შესახებ
CultureInfo^ CreateSpecificCulture(String^ *სახელი*) (სტატიკურია)	კულტურის დასახელებიდან ქმნის CultureInfo ობიექტს
array<CultureInfo^>^ GetCultures(CultureTypes *ტიპები*) (სტატიკურია)	გასცემს კულტურების სიას
CultureInfo^ ReadOnly(CultureInfo^ *ობიექტი*) (სტატიკურია)	გასცემს CultureInfo ობიექტის ვერსიას, რომლის მხოლოდ წაკითხვაა შესაძლებელი

ამ განყოფილებაში მოცემული ყველა პროგრამის დასაწყისში უნდა მოვათავსოთ დირექტივა:

```
using namespace System::Globalization;
```

CultureInfo კლასის თვისებები და ფუნქციები მოცემულია ცხრლებში 15.2 და 15.3.

მოცემული პროგრამით ხდება CultureInfo და მასთან დაკავშირებული კლასების გამოყენების დემონსტრირება.

```
{
//      ქართულ კულტურასთან მუშაობა
//      იქმნება ქართული კულტურა
CultureInfo^ QartuliKultura = gcnew CultureInfo(L"ka-GE");
label1->Text = QartuliKultura->NativeName;          //      გაიცემა - ქართული(საქართველო)
label2->Text = QartuliKultura->EnglishName;         //      გაიცემა - Georgian(Georgia)
//      ქართული კულტურისათვის განისაზღვრება თარიღისა და დროის ფორმატი
DateTimeFormatInfo^ TarigiDroisFormati = QartuliKultura->DateTimeFormat;
label5->Text += TarigiDroisFormati->LongDatePattern;
//      გაიცემა - yyyy'წლის'dd MM, dddd
//      ქართული კულტურისათვის განისაზღვრება ვალუტისა და რიცხვის ფორმატი
NumberFormatInfo^ RicxvisFormati = QartuliKultura->NumberFormat;
label3->Text = RicxvisFormati->CurrencySymbol;              //      გაიცემა - Lari
label4->Text = RicxvisFormati->NumberDecimalSeparator;      //      გაიცემა - ,
}
```

პროგრამაში იქმნება CultureInfo კლასის QartuliKultura ობიექტი, რომელიც ქართულ კულტურას წარმოადგენს. შემდეგ გაიცემა ამ კულტურის სახელი ქართულ და ინგლისურ ენებზე, თარიღის გრძელი ფორმატი, ვალუტის სიმბოლო და მთელისა და წილადის გამყოფი.

CurrentCulture და CurrentUICulture თვისებები. CurrentCulture თვისება გამოიყენება კულტურის ისეთი მონაცემის დაფორმატებისათვის, როგორიცაა რიცხვების, თარიღისა და დროის ფორმატი, ვალუტის სიმბოლო, დახარისხების რიგითობა და სტრიქონების შედარების წესები. CurrentUICulture თვისება გამოიყენება სამომხმარებლო ინტერფეისის ენისათვის. ის ახდენს შესაბამისი ფაილიდან კულტურაზე დამოკიდებული რესურსების, პირველ რიგში კი სამომხმარებლო ინტერფეისის ტექსტის მიღებას.მოცემული პროგრამით ხდება ამ თვისებებთან მუშაობის დემონსტრირება.

```
{
//      CurrentCulture და CurrentUICulture თვისებებთან მუშაობა
//      იქმნება ქართული კულტურა
Thread::CurrentThread->CurrentCulture = gcnew CultureInfo(L"ka-GE");
```

```
label1->Text = Thread::CurrentThread->CurrentCulture->Name + "\n";
//        იქმნება დიდი ბრიტანეთის კულტურა
CultureInfo^ InglisisKultura1 = gcnew CultureInfo(L"en-GB");
label1->Text += CultureInfo::CurrentCulture->Name + "\n";
//        სამომხმარებლო ინტერფეისისათვის მიმდინარე კულტურის დაყენება
//        მიმდინარე ნაკადისათვის განისაზღვრება იტალიის კულტურა
Thread::CurrentThread->CurrentUICulture = gcnew CultureInfo(L"it-IT");
label1->Text += Thread::CurrentThread->CurrentUICulture->Name + "\n";
//        იქმნება დიდი ბრიტანეთის კულტურა
CultureInfo^ InglisisKultura2 = gcnew CultureInfo(L"en-GB");
label1->Text += CultureInfo::CurrentUICulture->Name;
}
```

.NET გარემო ორ სხვადასხვა ნაკადში ორ სხვადასხვა კულტურასთან მუშაობს. როგორც პროგრამიდან ჩანს, მიმდინარე ნაკადისათვის მიმდინარე კულტურად განისაზღვრა საქართველო - "ka-GE", CultureInfo ობიექტის შექმნის გზით. ამ ობიექტზე მიმართვა ენიჭება Thread.CurrentThread.CurrentCulture თვისებას. შემდეგ იქმნება დამოუკიდებელი InglisisKultura1 ობიექტი, რომელიც განსაზღვრავს ინგლისის კულტურას - "en-GB". მაგრამ, ეს კულტურა არ არის მიმდინარე, ამიტომ ეკრანზე ისევ გამოჩნდება "ka-GE" კულტურა.

კულტურების ჩამონათვალი. CultureInfo ობიექტის GetCultures() ფუნქცია გასცემს კულტურების სიას. ფუნქციას პარამეტრად გადაეცემა CultureTypes პარამეტრი, რომელიც შეიცავს კულტურების ნაკრებს. 15.4 ცხრილში მოცემულია ამ პარამეტრის მნიშვნელობები.

ცხრილი 15.4. CultureTypes პარამეტრის მნიშვნელობები

მნიშვნელობა	აღწერა
AllCultures	ყველა კულტურა, ამოცნობილი .NET გარემოს მიერ
InstalledWin32Cultures	ყველა კულტურა, დაყენებული ოპერაციულ სისტემაში
NeutralCultures	ყველა ნეიტრალური კულტურა
SpecificCultures	ყველა სპეციფიკური კულტურა

მოცემული პროგრამით ხდება თითოეული კულტურის დასახელების ეკრანზე გამოტანა.

```
{
//        თითოეული კულტურის დასახელების ეკრანზე გამოტანა
for each ( CultureInfo^ Kultura in CultureInfo::GetCultures(CultureTypes::AllCultures) )
        listBox1->Items->Add(Kultura->EnglishName + "-" + Kultura->Name);
}
```

პროგრამაში listBox1 კომპონენტი გამოიყენება კულტურების სახელების გამოსატანად. ის შეგიძლიათ ჩასვათ Toolbox პანელიდან.

ინვარიანტული კულტურა. ინვარიანტული კულტურა (Invariant Language, Invariant Country) გამოიყენება იმ მონაცემებთან სამუშაოდ, რომლებიც არ იქნება ასახული რომელიმე კულტურის ელემენტების მიხედვით. ის დაკავშირებულია ინგლისურ ენასთან და არა რომელიმე ქვეყანასთან ან რეგიონთან. CultureInfo ობიექტი, რომელიც წარმოადგენს ინვარიანტულ კულტურას, შეგვიძლია ორი სხვადასხვა საშუალებით შევქმნათ:

```
CultureInfo^ InvariantuliKultura1 = gcnew CultureInfo("");
CultureInfo^ InvariantuliKultura2 = CultureInfo::InvariantCulture;
```

მოცემული პროგრამით ხდება თარიღისა და ვალუტის დაფორმატება საქართველოსა და დიდი ბრიტანეთის კულტურებისათვის.

```
{
```

```
//      თარიღისა და ვალუტის დაფორმატება
DateTime^ Tarigi = DateTime::Now;
Double   Valuta = 12345.67;
//      თარიღისა და ვალუტის დაფორმატება ქართული
//      კულტურის პარამეტრების შესაბამისად
CultureInfo^ SaqartvelosKultura = gcnew CultureInfo(L"ka-GE");
String^ QartuliTarigi = Tarigi->ToString(L"d", SaqartvelosKultura);
String^ QartuliValuta = Valuta.ToString(L"c", SaqartvelosKultura);
label1->Text = QartuliTarigi + "\n" + QartuliValuta + "\n";
//      თარიღისა და ვალუტის დაფორმატება ინგლისის
//      კულტურის პარამეტრების შესაბამისად
CultureInfo^ InglisisKultura = gcnew CultureInfo(L"en-GB");
String^ InglisuriTarigi = Tarigi->ToString(L"d", InglisisKultura);
String^ InglisuriValuta = Valuta.ToString(L"c", InglisisKultura);
label1->Text += InglisuriTarigi + "\n" + InglisuriValuta + "\n";
}
```

პროგრამაში გამოიყენება ToString() ფუნქციის გადატვირთული ვერსია, რომელიც აფორმატებს თარიღსა და ვალუტას.

გრაფიკა

.NET გარემოში განსაზღვრულია გრაფიკის კლასები, რომლებიც განკუთვნილია GDI+ ინტერფეისთან სამუშაოდ (Graphic Device Interface, გრაფიკული მოწყობილობების ინტერფეისი). ეს კლასები საშუალებას გვაძლევს დავხატოთ გეომეტრიული ფიგურები, ვიმუშაოთ გამოსახულების შემცველ ფაილებთან და ტიპოგრაფიულ შრიფტებთან. GDI+ არის კლასების ბიბლიოთეკა, რომლის დანიშნულებაა გრაფიკული ობიექტების ასახვა ჩვენი კომპიუტერის აპარატურული საშუალებებით. მისი მიზანია პროგრამა-დანართების შემმუშავებლების იზოლირება გრაფიკული მოწყობილობის რეალიზების დეტალებისაგან. ჩვენ კოდს შეუძლია გამოიძახოს GDI+ ბიბლიოთეკაში შემავალი კლასები. ის დაინსტალდება Windows XP, Windows .NET Server ან .NET გარემოს ინსტალაციისას. GDI+ კლასები სამ ფუნქციას ასრულებს:

- ვექტორული გრაფიკის ისეთი ელემენტების ასახვა, როგორიცაა წერტილები, ხაზები და გეომეტრიული ფიგურები.
- გრაფიკული ობიექტების შემცველი ფაილების ასახვა, მაგალითად, bmp, gif, tif, jpg და png ფორმატის მქონე.
- შრიფტების ასახვა.

ცხრილი 15.5. Graphics კლასის თვისებები

თვისება	ტიპი	აღწერა
Clip	Region^	განსაზღვრავს ხატვის უბანს
ClipBounds	RectangleF	შეიცავს Rectangle ობიექტს, რომელიც განსაზღვრავს ხატვის უბანს
CompositingMode	CompositingMode	შედგენილი გამოსახულებებისათვის მიუთითებს ხატვის რეჟიმს
CompositingQuality	CompositingQuality	შეიცავს გამოსახულებების ხარისხს
DpiX	float	განსაზღვრავს ჰორიზონტალური გარჩევის უნარს Graphics ობიექტისთვის
DpiY	float	განსაზღვრავს ვერტიკალური გარჩევის უნარს Graphics ობიექტისთვის
InterpolatingMode	InterpolatingMode	განსაზღვრავს შუალედური წერტილების გამოთვლის რეჟიმს
IsClipEmpty	bool	მიუთითებს, არის თუ არა კვეთის უბანი ცარიელი
PageScale	float	შეიცავს მასშტაბირების კოეფიციენტს Graphics ობიექტისთვის
PageUnit	GraphicsUnit	განსაზღვრავს გამოყენებულ ზომის ერთეულს
PixelOffsetMode	PixelOffsetMode	განსაზღვრავს პიქსელების წანაცვლების საშუალებას ხატვისას
RenderingOrigin	Point	განსაზღვრავს კოორდინატების დასაწყისს ფუნჯისა და შტრიხებისათვის
SmoothingMode	SmoothingMode	მართავს ხაზის სიგლუვეს Graphics ობიექტზე მათი ხატვისას
TextContrast	int	განსაზღვრავს კონტრასტის მნიშვნელობას ტექსტისათვის
TextRenderingHint	TextRenderingHint	განსაზღვრავს შემხსენებელ შეტყობინებას ტექსტისათვის
Transform	Matrix^	განსაზღვრავს გეომეტრიულ გარდაქმნას
VisibleClipbounds	RectangleF	შეიცავს კვეთის ხილულ უბანს Graphics ობიექტისთვის

ცხრილი 15.6. Graphics კლასის ფუნქციები

ფუნქცია	აღწერა
void AddMetafileComment(array<Byte>^ *კომენტარი*)	Metafile ობიექტს უმატებს კომენტარს
GraphicsContainer^ BeginContainer()	ხსნის ახალ გრაფიკულ კონტეინერს
void Clear(Color *ფერი*)	ასუფთავებს სახატავ ზედაპირს
void DrawArc(Pen^ *კალამი,* Rectangle *სწორკუთხედი,* float *საწყისი_კუთხე,* float *მოღუნვის_კუთხე*)	ხატავს რკალს

ცხრილი 15.6. (გაგრძელება)

void DrawBezier(Pen^ *კალამი*, float *x1*, float *y1*, float *x2*, float *y2*, float *x3*, float *y3*, float *x4*, float *y4*)	ხატავს ბაზიეს მრუდს
void DrawBeziers(Pen^ *კალამი*, array<Point>^ *წერტილები*)	ხატავს ბაზიეს მრუდების სერიას
void DrawClosedCurve(Pen^ *კალამი*, array<Point>^ *წერტილები*)	ხატავს დახურულ ფუნდამენტურ მრუდს
void DrawCurve(Pen^ *კალამი*, array<Point>^ *წერტილები*)	ხატავს ფუნდამენტურ მრუდს
void DrawEllipse(Pen^ *კალამი,* Rectangle *სწორკუთხედი*)	ხატავს ელიფსს
void DrawIcon(Icon^ *პიქტოგრამა*, Rectangle *სწორკუთხედი*)	ხატავს პიქტოგრამას
void DrawIconUnstretched(Icon^ *პიქტოგრამა*, Rectangle *სწორკუთხედი*)	ხატავს პიქტოგრამას მასშტაბირების გარეშე
DrawImage()	ხატავს გამოსახულებას
DrawImageUnscaled()	ხატავს გამოსახულებას მასშტაბირების გარეშე
void DrawLine(Pen^ *კალამი,* float x1, float y1, float x2, float y2)	ხატავს ხაზს
void DrawLines(Pen^ *კალამი*, array<Point>^ *წერტილები*)	ხატავს ხაზების სერიას
void DrawPath(Pen^ *კალამი*, GraphicsPath^ *ობიექტი*)	ხატავს GraphicsPath ობიექტს
void DrawPie(Pen^ *კალამი,* Rectangle *სწორკუთხედი,* float *საწყისი_კუთხე,* float *მოლუნვის_კუთხე*)	ხატავს სექტორს
void DrawPolygon(Pen^ *კალამი*, array<Point>^ *წერტილები*)	ხატავს მრავალკუთხედს
void DrawRectangle(Pen^ *კალამი,* Rectangle *მართკუთხედი*)	ხატავს მართკუთხედს
void DrawRectangles(Pen^ *კალამი*, array<Rectangle>^ *მართკუთხედები*)	ხატავს მართკუთხედების სერიას
void DrawString(String^ *სტრიქონი,* Font^ *შრიფტი,* Brush^ *ფუნჯი,* float *x*, float *y*)	ხატავს ტექსტის სტრიქონს
void EndContainer(GraphicsContainer^ *კონტეინერი*)	ხურავს გრაფიკულ კონტეინერს
EnumerateMetafile()	ახდენს მეტაფაილის ვიზუალიზებას
void ExcludeClip(Rectangle *მართკუთხედი*)	მართკუთხედს უმატებს კვეთის უბანს

ცხრილი 15.6. (გაგრძელება)

void FillClosedCurve(Brush^ *ფუნჯი*, array<Point>^ *წერტილები*)	ფერით ავსებს ჩაკეტილ მრუდს
void FillEllipse(Brush^ *ფუნჯი*, Rectangle *მართკუთხედი*)	ფერით ავსებს ელიფსს
void FillPath(Brush^ *ფუნჯი*, GraphicsPath^ *ობიექტი*)	ფერით ავსებს GraphicsPath ობიექტს
void FillPie(Brush^ *ფუნჯი*, Rectangle *მართკუთხედი*, float *საწყისი_კუთხე*, float *მოღუნვის_კუთხე*)	ფერით ავსებს სექტორს
void FillPolygon(Brush^ *ფუნჯი*, array<Point>^ *წერტილები*)	ფერით ავსებს მრავალკუთხედს
void FillRectangle(Brush^ *ფუნჯი*, Rectangle *მართკუთხედი*)	ფერით ავსებს მართკუთხედს
void Fill Rectangles(Brush^ *ფუნჯი*, array<Rectangle>^ *მართკუთხედები*)	ფერით ავსებს მართკუთხედების სერიას
void FillRegion(Brush^ *ფუნჯი*, Region^ *ობიექტი*)	ფერით ავსებს Region ობიექტს
void Flush()	იწვევს რიგში მყოფი ოპერაციების იძულებით შესრულებას
Graphics^ FromImage(Image^ *ობიექტი*) (სტატიკურია)	Image ობიექტიდან ქმნის Graphics ობიექტს
Color GetNearestColor(Color *ფერი*)	გასცემს მითითებულ ფერთან ახლოს მყოფ ფერს
void IntersectClip(Rectangle *მართკუთხედი*)	ცვლის კვეთის უბანს მართკუთხედთან მისი გადაკვეთის გზით
bool IsVisible(Point *წერტილი*)	გასცემს true-ს თუ მითითებული წერტილი ჩანს, წინააღმდეგ შემთხვევაში კი - false-ს
array<Region^>^ MeasureCharacterRanges(String^ *სტრიქონი*, Font^ *შრიფტი*, RectangleF *ადგილი_ზომა*, StringFormat^ *სტრიქონის_ფორმატი*)	გასცემს ინფორმაციას ტექსტური სტრიქონის შესახებ
SizeF MeasureString(String^ *სტრიქონი*, Font^ *შრიფტი*)	გასცემს სტრიქონის ზომას, თუ ის დახატულია მითითებული შრიფტით
void ResetClip()	აუქმებს კვეთის უბანს
void ResetTransform()	აუქმებს Graphics ობიექტის გარდაქმნას
void Restore(GraphicsState^ *სტატუსი*)	აღადგენს GraphicsState ობიექტში შენახულ მდგომარეობას
void RotateTransform(float *კუთხე*)	ახდენს Graphics ობიექტის მობრუნებას
GraphicsState^ Save()	მდგომარეობას ინახავს GraphicsState ობიექტში
void SetClip(Rectangle *მართკუთხედი*)	აყენებს კვეთის უბანს

ცხრილი 15.7. Pen კლასის თვისებები

თვისება	ტიპი	აღწერა
Alignment	PenAlignment	შეიცავს Pen ობიექტის საზღვარზე გასწორების სახეს
Brush	Brush^	შეიცავს Brush ობიექტს, რომელსაც იყენებს მოცემული Pen ობიექტი
Color	Color	შეიცავს Pen ობიექტის ფერს
CustomEndCap	CustomLineCap^	შეიცავს ხაზის დაბოლოების სტილს
CustomStartCap	CustomLineCap^	შეიცავს ხაზის დასაწყისის სტილს
DashCap	DashCap	შეიცავს პუნქტირის მონაკვეთების დაბოლოებების სტილს
DashOffset	float	შეიცავს მანძილს ხაზის დასაწყისიდან პუნქტირის შაბლონამდე
DashPattern	array<float>^	პუნქტირის მონაკვეთებისა და მათ შორის მანძილების მასივია
DashStyle	DashStyle	შეიცავს პუნქტირის მონაკვეთების სტილს
EndCap	LineCap	შეიცავს ხაზის დაბოლოების სტილს
LineJoin	LineJoin	შეიცავს რამდენიმე ხაზს შორის შეერთების სტილს
MiterLimit	float	შეიცავს ჩამოჭრილი კუთხეების სისქეს რამდენიმე ხაზის შეერთებისას
PenType	PenType	შეიცავს სტილს, რომელიც გამოიყენება Pen ობიექტის მიერ ხაზების გავლებისას
StartCap	LineCap	შეიცავს ხაზის დასაწყისის სტილს
Width	float	ხაზის სისქეა პიქსელებში

ცხრილი 15.8. Pen კლასის ფუნქციები

ფუნქცია	აღწერა
Object^ Clone()	ქმნის Pen ობიექტის ასლს
void ResetTransform()	ახდენს გარდაქმნის დაყვანას საწყის მდგომარეობაზე
void RotateTransform(float *კუთხე*)	ახდენს გარდაქმნის მობრუნებას
void ScaleTransform(float x, float y)	ახდენს გარდაქმნის მასშტაბირებას
void TranslateTransform(float x, float y)	ახდენს გარდაქმნის ტრანსლაციას

Graphics კლასი განსაზღვრავს სახატავ უბანს (ფურცელს, ტილოს), Pen კლასი გამოიყენება ხაზებისა და გრაფიკული ფიგურების დასახატად, Brush კლასი გამოიყენება ფიგურების შესავსებად არჩეული ფერით. Rectangle კლასი განსაზღვრავს სახატავი უბნის საზღვრებს. Graphics კლასის თვისებები და ფუნქციები მოცემულია 16.5 და 16.6 ცხრილებში. Pen კლასის თვისებები და ფუნქციები მოცემულია 16.7 და 16.8 ცხრილებში.

სიგრძის ერთეული ყველა გრაფიკული ობიექტისათვის არის პიქსელი. კოორდინატების დასაწყისად ითვლება Graphics ობიექტის ზედა მარცხენა კუთხე.

Graphics ობიექტი წარმოადგენს ხატვის ზედაპირს და შეგვიძლია განვიხილოთ როგორც სახატავი ფურცელი (ტილო). Pen ობიექტი შეიცავს დასახატი ხაზის თვისებებს.

გეომეტრიული ფიგურის ხატვა

განვიხილოთ Graphics კლასის ზოგიერთი ფუნქცია.

Graphics კლასის ობიექტი იქმნება CreateGraphics() ფუნქციის საშუალებით. მაგალითად,

```
Graphics^ SaxataviPurceli = CreateGraphics();
```

DrawArc() ფუნქცია გამოიყენება ელიფსის ნაწილის (რკალის) დასახატად. მისი სინტაქსია:

void *graphics*.DrawArc(Pen^ *ფუნჯი,* Rectangle *მართკუთხედი,*
float *საწყისი_კუთხე,* float *მოღუნვის_კუთხე*);

აქ ***graphics*** არის Graphics ტიპის ობიექტი. მაგალითი:

```
{
//      სახატავი ფურცლის შექმნა
Graphics^ SaxataviPurceli = CreateGraphics();
//      კალმის შექმნა
Pen^ Kalami = gcnew Pen(Color::Red, 2);
//      ხატვის უბნის შექმნა
Rectangle Otxkutxedi(0, 0, 200, 300);
//      რკალის ხატვა
SaxataviPurceli->DrawArc(Kalami, Otxkutxedi, 0, 90);
}
```

როგორც პროგრამიდან ჩანს, ჯერ იქმნება სახატავი ფურცელი. შემდეგ იქმნება წითელი ფერის კალამი, რომლის სისქეა 2 პიქსელი. შემდეგ იქმნება ხატვის უბანი, რომელშიც უნდა მოთავსთდეს რკალი. ამ უბნის საწყისი წერტილია (0,0), საბოლოო წერტილი კი - (200,300). ბოლოს სრულდება რკალის ხატვა.

DrawRectangle() ფუნქცია გამოიყენება მართკუთხედის დასახატად. მისი სინტაქსია:

void *graphics*.DrawRectangle(Pen^ *ფუნჯი,* Rectangle *მართკუთხედი*);

მაგალითი:

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
Pen^ Kalami = gcnew Pen(Color::Red, 2);
Rectangle Otxkutxedi(0, 0, 200, 300);
SaxataviPurceli->DrawRectangle(Kalami, Otxkutxedi);
}
```

DrawEllipse() ფუნქცია გამოიყენება ელიფსის დასახატად. მისი სინტაქსია:

void *graphics*.DrawEllipse(Pen^ *ფუნჯი,* Rectangle *მართკუთხედი*);

მაგალითი:

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
Pen^ Kalami = gcnew Pen(Color::Red, 2);
Rectangle Otxkutxedi(0, 0, 200, 300);
SaxataviPurceli->DrawEllipse(Kalami, Otxkutxedi);
}
```

DrawPie() ფუნქცია გამოიყენება სექტორის დასახატად. მისი სინტაქსია:

void *graphics*.DrawPie(Pen^ *ფუნჯი,* Rectangle *მართკუთხედი,*
float *საწყისი_კუთხე,* float *მოღუნვის_კუთხე*);

მაგალითი:

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
```

```
Pen^ Kalami = gcnew Pen(Color::Red, 2);
Rectangle Otxkutxedi(0, 0, 200, 300);
SaxataviPurceli->DrawPie(Kalami, Otxkutxedi, 0, 90);
}
```

DrawLine() ფუნქცია გამოიყენება ხაზის გასავლებად საწყისი წერტილიდან საბოლოო წერტილამდე. მისი სინტაქსია:

void *graphics*.DrawLine(Pen^ *ფუნჯი,* float x1, float y1, float x2, float y2);

აქ x1 და y1 არის საწყისი წერტილის კოორდინატები, ხოლო x2 და y2 კი - საბოლოო წერტილის. მაგალითი:

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
Pen^ Kalami = gcnew Pen(Color::Red, 2);
Rectangle Otxkutxedi(0, 0, 200, 300);
SaxataviPurceli->DrawLine(Kalami, 0, 0, 100, 100);
}
```

DrawPolygon() ფუნქცია გამოიყენება მრავალკუთხედის ასაგებად. მისი სინტაქსია:

void *graphics*.DrawPolygon(Pen^ *ფუნჯი*, array<Point>^ *წერტილები*);

მაგალითი:

```
{
//      წერტილების შექმნა, რომლებიც მრავალკუთხედს განსაზღვრავენ
Point wertili1(50, 50);
Point wertili2(100, 25);
Point wertili3(130, 5);
Point wertili4(170, 50);
Point wertili5(210, 100);
Point wertili6(50, 50);
array<Point>^ WertilebisMasivi = gcnew array<Point> (6)
                                        { wertili1, wertili2, wertili3, wertili4, wertili5, wertili6 };
Graphics^ SaxataviPurceli = CreateGraphics();
Pen^ Kalami = gcnew Pen(Color::Red, 2);
SaxataviPurceli->DrawPolygon(Kalami, WertilebisMasivi);
}
```

DrawLines() ფუნქცია გამოიყენება ხაზების გასავლებად საწყისი წერტილიდან საბოლოო წერტილამდე. მისი სინტაქსია:

void *graphics*.DrawLines(Pen^ *ფუნჯი*, array<Point>^ *წერტილები*);

მაგალითი:

```
{
//      წერტილების შექმნა ხაზებისათვის
Point wertili1(50, 50);
Point wertili2(100, 25);
Point wertili3(130, 5);
Point wertili4(170, 50);
Point wertili5(210, 100);
Point wertili6(50, 50);
array<Point>^ WertilebisMasivi = gcnew array<Point> (6)
                                        { wertili1, wertili2, wertili3, wertili4, wertili5, wertili6 };
Graphics^ SaxataviPurceli = CreateGraphics();
```

```
Pen^ Kalami = gcnew Pen(Color::Red, 2);
SaxataviPurceli->DrawLines(Kalami, WertilebisMasivi);
}
```

Pen კლასის კონსტრუქტორს აქვს 4 გადატვირთული ვერსია. პირველი ვერსიის სინტაქსია:

Pen^ *ობიექტის_სახელი* = gcnew Pen(Brush *ფუნჯი*);

მაგალითი:

```
{
Brush^ PunjiSolid = gcnew SolidBrush(Color::Red);
Pen^ Kalami = gcnew Pen(PunjiSolid);
}
```

მეორე ვერსიის სინტაქსია:

Pen^ *ობიექტის_სახელი* = gcnew Pen(Color *ფერი*);

მაგალითი:

```
{
Pen^ Kalami = gcnew Pen(Color::Red);
}
```

მესამე ვერსიის სინტაქსია:

Pen^ *ობიექტის_სახელი* = gcnew Pen(Brush *ფუნჯი,* float *სიგანე*);

მაგალითი:

```
{
Brush^ PunjiSolid = gcnew SolidBrush(Color::Red);
Pen^ Kalami = gcnew Pen(PunjiSolid, 3);
}
```

მეოთხე ვერსიის სინტაქსია:

Pen^ *ობიექტის_სახელი* = gcnew Pen(Color *ფერი,* float *სიგანე*);

მაგალითი:

```
{
Pen^ Kalami = gcnew Pen(Color::Red, 3);
}
```

მოცემული პროგრამით ხდება ხაზისა და ისრის გავლების დემონსტრირება.

```
{
int x1 = 5, y1 = 5, x2 = 200, y2 = 200;
Graphics^ SaxataviPurceli = CreateGraphics();
Pen^ Kalami = gcnew Pen(Color::Red, 5);
SaxataviPurceli->DrawLine(Kalami, x1, y1, x2, y2);
//
x1 = 5; y1 = 200; x2 = 200; y2 = 5;
//      ხაზის დასაწყისს ექნება ისრის ფორმა
Kalami->StartCap  = Drawing2D::LineCap::ArrowAnchor;
//      ხაზის დაბოლოება იქნება მომრგვალებული
Kalami->EndCap   = Drawing2D::LineCap::Round;
SaxataviPurceli->DrawLine(Kalami, x1, y1, x2, y2);
}
```

პროგრამაში ჯერ იქმნება Graphics კლასის SaxataviPurceli ობიექტი. შემდეგ იქმნება Pen კლასის Kalami ობიექტი, რომელსაც ექნება წითელი ფერი, სისქე კი - 5 პიქსელი. თუ სისქე არ არის მითითებული, მაშინ ის აიღება 1-ის ტოლი. DrawLine ფუნქციი (0,0) წერტილიდან

გაავლებს წითელ ხაზს (200, 200) წერტილამდე. შემდეგ პირველი წერტილის კოორდინატი ხდება (5, 200), მეორე წერტილის კი - (200, 5). ხაზის დასაწყისს ექნება ისრის ფორმა, ბოლოს კი - მომრგვალებული სახე. ამის შემდეგ, ისარი გაივლება (5, 200) წერტილიდან (200, 5) წერტილამდე.

ფიგურის შევსება ფერით

Brush კლასი გამოიყენება სხვადასხვა ფერით და შაბლონით ფიგურების შესავსებად. ის აბსტრაქტულია და ამიტომ ამ კლასის ობიექტებს ვერ შევქმნით. მისგან წარმოებული კლასები მოცემულია 15.9 ცხრილში.

ცხრილი 15.9. Brush კლასიდან წარმოებული კლასები

კლასი	გამოყენება
HatchBrush	გამოიყენება უბნის შესავსებად სტანდარტული შაბლონების ნაკრებიდან ამორჩეული შაბლონის მიხედვით
LinearGradientBrush	გამოიყენება უბნის შესავსებად ორ ფერს შორის თანაბარი გადასვლით სწორი ხაზის გასწვრივ
PathGradientBrush	გამოიყენება უბნის შესავსებად ორ ფერს შორის თანაბარი გადასვლით ნებისმიერი ხაზის გასწვრივ
SolidBrush	გამოიყენება უბნის შესავსებად ერთტონიანი ფერით
TextureBrush	გამოიყენება უბნის შესავსებად ფაილიდან წაკითხული გამოსახულებით

მოცემული პროგრამით ხდება სხვადასხვა გეომეტრიული ფიგურის სხვადასხვა სტილით შევსების დემონსტრირება, კერძოდ, მართკუთხედისთვის გამოიყენება გრადიენტული შევსება, ელიფსისთვის - შაბლონის მიხედვით შევსება, ხოლო სექტორისთვის კი - თანაბარი ფერით შევსება.

```
{
//      სხვადასხვა გეომეტრიულ ფიგურის სხვადასხვა სტილით შევსება
int x = 0;
int y = 0;
int Sigane  = 200;
int Simagle = 300;
//      სახატავი ფურცლის შექმნა
Graphics^ SaxataviPurceli = CreateGraphics();
//      სახატავი უბნის შექმნა
Rectangle Otxkutxedi(x, y, Sigane, Simagle);
//      ფუნჯის შექმნა თანაბარი ფერით შევსებისათვის
Brush^ PunjiSolid = gcnew SolidBrush(Color::Red);
//      ფუნჯის შექმნა შაბლონის მიხედვით შევსებისათვის
Brush^ PunjiHatch = gcnew HatchBrush(HatchStyle::Horizontal, Color::Blue, Color::Yellow);
//      ფუნჯის შექმნა გრადიენტული შევსებისათვის
Brush^ PunjiGradient = gcnew LinearGradientBrush(Otxkutxedi, Color::Black, Color::Aqua, 45, false);
//      ოთხკუთხედის, ელიფსისა და სექტორის შევსება ფერით
SaxataviPurceli->FillRectangle(PunjiGradient, Otxkutxedi);
SaxataviPurceli->FillEllipse(PunjiHatch, 5, 110, 100, 100);
SaxataviPurceli->FillPie(PunjiSolid, 60, 60, 120, 120, 270, 90);
```

```
}
```

სტრიქონის ხატვა

DrawString() ფუნქცია გამოიყენება სტრიქონების დასახატად. მისი სინტაქსია:

void *graphics*.DrawString(String^ *სტრიქონი,* Font^ *შრიფტი,* Brush^ *ფუნჯი,* float *x*, float *y*);

მოცემული პროგრამით ხდება ჰორიზონტალური სტრიქონის ხატვის დემონსტრირება.

```
{
float x = 10.0f;
float y = 10.0f;
String^ DasaxatiStriqoni = L"ანა და საბა";
//	სახატავი ფურცლის შექმნა
Graphics^ SaxataviPurceli = CreateGraphics();
//	შრიფტის განსაზღვრა სტრიქონისთვის
System::Drawing::Font^ Shrifti = gcnew System::Drawing::Font(L"Sylfaen", 14);
//	ფუნჯის შექმნა
SolidBrush^ Punji = gcnew SolidBrush(Color::Black);
//	ჰორიზონტალური სტრიქონის ხატვა
SaxataviPurceli->DrawString(DasaxatiStriqoni, Shrifti, Punji, x, y);
}
```

მოცემული პროგრამით ხდება ვერტიკალური სტრიქონის ხატვის დემონსტრირება.

```
{
float x = 10.0f;
float y = 10.0f;
String^ DasaxatiStriqoni = L"საბა და ანა";
//	სახატავი ფურცლის შექმნა
Graphics^ SaxataviPurceli = CreateGraphics();
//	შრიფტის განსაზღვრა სტრიქონისთვის
System::Drawing::Font^ Shrifti = gcnew System::Drawing::Font(L"Sylfaen", 14);
//	ფუნჯის შექმნა
SolidBrush^ Punji = gcnew SolidBrush(Color::Black);
//	სტრიქონისათვის ვერტიკალური მიმართულების არჩევა
StringFormat^ StriqonisFormati = gcnew StringFormat(StringFormatFlags::DirectionVertical);
//	ვერტიკალური სტრიქონის ხატვა
SaxataviPurceli->DrawString(DasaxatiStriqoni, Shrifti, Punji, x, y, StriqonisFormati);
}
```

გამოსახულებასთან მუშაობა

გამოსახულებასთან სამუშაოდ გამოიყენება Bitmap ობიექტი. ის მუშაობს bmp, gif, jpg, png და tif ფაილებთან. GDI+ ინტერფეისი ავტომატურად ცვლის გამოსახულების ზომას ისე, რომ ის მოთავსთდეს ხატვის უბანში. 15.10 და 15.11 ცხრილებში მოცემულია Bitmap კლასის თვისებები და ფუნქციები.

ცხრილი 15.10. Bitmap კლასის თვისებები

თვისება	ტიპი	აღწერა
Flags	int	ალამი-ატრიბუტების ნაკრები
FrameDimensionsList	array<Guid>^	მასივი, რომელიც მიუთითებს გამოსახულებაში კადრების განზომილებებს
Height	int	გამოსახულების სიმაღლე მასშტაბირების გარეშე
HorizontalResolution	float	ჰორიზონტალური გარჩევის უნარი პიქსელი/დუიმზე
Palette	ColorPalette^	ფერების პალიტრა
PhysicalDimension	SizeF	გამოსახულების სიგანე და სიმაღლე
PixelFormat	PixelFormat	გამოსახულების პიქსელების ფორმატი
PropertyIDList	array<int>^	თვისებების იდენტიფიკატორების მასივი, რომლებიც გამოსახულებასთან ერთადაა შენახული
PropertyItems	array <PropertyItem^>^	PropertyItem ობიექტების მასივი, რომელიც მოცემულ გამოსახულებას აღწერს
RawFormat	ImageFormat^	გამოსახულების ფორმატი
Size	Size	გამოსახულების სიგანე და სიმაღლე
VerticalResolution	float	ვერტიკალური გარჩევის უნარი პიქსელი/დუიმზე
Width	int	გამოსახულების სიგანე მასშტაბირების გარეშე

Bitmap კლასის კონსტრუქტორს აქვს 12 გადატვრთული ვერსია. ჩვენ მოვიყვანთ ორი მათგანის სინტაქსს:

Bitmap(String^ *ფაილის_სახელი*);

Bitmap(int *სიგანე*, int *სიმაღლე*);

მოცემული პროგრამით ხდება მითითებული ფაილიდან გამოსახულების ასახვის დემონსტრირება.

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
Bitmap^ Gamosaxuleba2 = gcnew Bitmap("Surati.JPG");
SaxataviPurceli->DrawImage(Gamosaxuleba2, 5, 5, 250, 250);
}
```

მოცემული პროგრამით ხდება დიალოგურ ფანჯარაში არჩეული ფაილიდან გამოსახულების ასახვის დემონსტრირება.

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
OpenFileDialog^ Surati_Failidan = gcnew OpenFileDialog();
Bitmap^ Gamosaxuleba;
if ( Surati_Failidan->ShowDialog() == System::Windows::Forms::DialogResult::OK )
{
Gamosaxuleba = gcnew Bitmap(Surati_Failidan->FileName);
SaxataviPurceli->DrawImage(Gamosaxuleba, 5, 5, 250, 250);
}
}
```

ცხრილი 15.11. Bitmap კლასის ფუნქციები

ფუნქცია	აღწერა
Object^ Clone() (სტატიკური)	ქმნის Bitmap ობიექტის ასლს
Bitmap^ FromHicon(IntPtr *პიქტოგრამის_დესკრიპტორი*)	ქმნის Bitmap ობიექტს Windows-ის პიქტოგრამის დესკრიპტორიდან
RectangleF GetBounds(GraphicsUnit% *უბანი*)	გასცემს მართკუთხა უბანს, რომელშიც უნდა მოთავსთდეს გამოსახულება
int GetFrameCount(FrameDimension^ *განზომილება*)	გასცემს გამოსახულებაში კადრების რაოდენობას
IntPtr GetHicon()	გასცემს Windows-ის პიქტოგრამის დესკრიპტორს მოცემული გამოსახულებისათვის
Color GetPixel(int x, int y)	გასცემს გამოსახულების მითითებული პიქსელის ფერს
void MakeTransparent()	გამოსახულების ნაწილებს გამჭვირვალეს ხდის
void RotateFlip(RotateFlipType *ტიპი*)	გამოსახულებს აბრუნებს ან სარკისებურად ასახავს
int SelectActiveFrame(FrameDimension^ *განზომილება*, int *კადრის_ინდექსი*)	მრავალკადრიანი გამოსახულებიდან ირჩევს ერთ კადრს
void SetPixel(int *x*, int *y*, Color *ფერი*)	აყენებს გამოსახულების მითითებული პიქსელის ფერს
void SetResolution(float xDpi, float yDpi)	აყენებს გამოსახულების გარჩევის უნარს

SetPixel() ფუნქცია გამოიყენება მითითებული პიქსელისთვის (მონიტორის ეკრანის წერტილისთვის) ფერის მისანიჭებლად. მისი სინტაქსია:

void SetPixel(int x, int y, Color *ფერი*);

მოცემული პროგრამით ხდება SetPixel() ფუნქციის გამოყენებით გამოსახულების ასახვის დემონსტრირება.

```
{
Graphics^ SaxataviPurceli = CreateGraphics();
Bitmap^ Gamosaxuleba1 = gcnew Bitmap(90, 90);                    // სიგანე = სიმაღლე = 90
//        თეთრი ოთხკუთხედის ხატვა
for ( int x = 0; x < Gamosaxuleba1->Height; ++x )
   for ( int y = 0; y < Gamosaxuleba1->Width; ++y )
                Gamosaxuleba1->SetPixel(x, y, Color::White);
//        წითელი ხაზის ხატვა
for ( int x = 0; x < Gamosaxuleba1->Height; ++x )
Gamosaxuleba1->SetPixel(x, x, Color::Red);
//        გამოსახულების გამოტანა
SaxataviPurceli->DrawImage(Gamosaxuleba1, 0, 0, 100, 100);
}
```

მონაცემების დაშიფვრა და გაშიფვრა

მონაცემების დაცვის ერთ-ერთი სახეა მათი დაშიფვრა. .NET გარემო შეიცავს კლასებს, რომლებიც უზრუნველყოფენ როგორც სიმეტრიულ, ისე ასიმეტრიულ კრიპტოგრაფიას. სიმეტრიულ კრიპტოგრაფიას ზოგჯერ უწოდებენ კრიპტოგრაფიას დახურული გასაღებით, ხოლო ასიმეტრიულ კრიპტოგრაფიას - კრიპტოგრაფიას ღია გასაღებით.

სიმეტრიული კრიპტოგრაფიის შემთხვევაში მონაცემების დაშიფვრისა და გაშიფვრისათვის ერთი და იგივე გასაღები გამოიყენება. .NET გარემოში გამოყენებულია სიმეტრიული კრიპტოგრაფიის ოდნავ გართულებული ფორმა, რომელსაც დაშიფრული ბლოკების გადაბმა ეწოდება. მონაცემების გასაშიფრად ამ ფორმის გამოყენების შემთხვევაში უნდა ვიცოდეთ არა მხოლოდ გასაღები, არამედ მაინიციალებელი ვექტორიც. სიმეტრიული კრიპტოგრაფიის მიმართულებით შექმნილია ბევრი ალგორითმი, რომლებიც ორიენტირებულია იმაზე, რომ რთული იყოს გასაღების გარეშე მონაცემების გაშიფვრა. .NET გარემოში გამოყენებულია სიმეტრიული კრიპტოგრაფიის ოთხი ალგორითმი: DES, RC2, Rijndael და Triple DES.

ასიმეტრიული კრიპტოგრაფიის შემთხვევაში მონაცემების დაშიფვრისა და გაშიფვრისათვის გამოიყენება ორი გასაღები: ღია და დახურული. ღია გასაღების გამოყენებით დაშიფრული მონაცემების გაშიფვრა შესაძლებელია მხოლოდ შესაბამისი დახურული გასაღებით. .NET გარემოში გამოყენებულია ასიმეტრიული კრიპტოგრაფიის ორი ალგორითმი: RSA და DSA.

თუ გასაღები არ გვაქვს, მაშინ ასიმეტრიული შიფრის "გატეხვა" უფრო რთულია, ვიდრე სიმეტრიულის. "გატეხვა" არის პროცესი, როდესაც გასაღები არ გვაქვს და სხვადასხვა ხერხით ვცდილობთ დაშიფრული მონაცემების გაშიფვრას. გარდა ამისა, მონაცემების ასიმეტრიული დაშიფვრა მოითხოვს უფრო მეტ გამოთვლებს, ვიდრე სიმეტრიული. კრიპტოგრაფიის კლასები მოთავსებულია System::Security::Cryptography სახელების სივრცეში.

სიმეტრიული კრიპტოგრაფია

სიმეტრიული კრიპტოგრაფიისათვის გამოიყენება TripleDESCryptoServiceProvider კლასი. მისი თვისებები მოცემულია 15.12 ცხრილში, ფუნქციები კი - 15.13 ცხრილში.

მონაცემების სიმეტრიული დაშიფვრისათვის გამოიყენება ნაკადები. დაშიფვრა ხორციელდება CryptoStream ობიექტის საშუალებით. რადგან ერთი ნაკადის გამოსასვლელი შეიძლება მივაწოდოთ მეორე ნაკადის შესასვლელს, ამიტომ CryptoStream კლასი შეგვიძლია გამოვიყენოთ FileStream კლასთან ერთად ფაილში დაშიფრული მონაცემების ჩასაწერად და წასაკითხად, აგრეთვე, NetworkStream კლასთან ერთად ქსელში გადასაცემი მონაცემების დასაშიფრად.

მოცემული პროგრამით ხდება ფაილში ჩასაწერი მონაცემების დაშიფვრა.

```
{
//      ფაილში ჩასაწერი მონაცემების დაშიფვრა
FileStream^ SafailoNakadi1 = File::Create("temp.txt");
//      ახალი კრიპტოგრაფიული პროვაიდერის შექმნა
TripleDESCryptoServiceProvider^ CriptoProvaideri = gcnew TripleDESCryptoServiceProvider();
//      დამშიფრავი ნაკადის შექმნა საფაილო ნაკადში
CryptoStream^ CriptoNakadi =
gcnew CryptoStream(SafailoNakadi1, CriptoProvaideri->CreateEncryptor(), CryptoStreamMode::Write);
StreamWriter^ ChawerisNakadi = gcnew StreamWriter(CriptoNakadi);
ChawerisNakadi->WriteLine(L"საბა და ანა");
ChawerisNakadi->WriteLine(L"ლიკა და რომანი");
```

```
ChawerisNakadi->Close();
//          გასაღებისა და მაინიციალიზებელი ვექტორის შენახვა
FileStream^ SafailoNakadi2  = File::Create("encrypted.key");
BinaryWriter^ OrobitiNakadi = gcnew BinaryWriter(SafailoNakadi2);
OrobitiNakadi->Write(CriptoProvaideri->Key);
OrobitiNakadi->Write(CriptoProvaideri->IV);
OrobitiNakadi->Close();
}
```

ცხრილი 15.12. TripleDESCryptoServiceProvider კლასის თვისებები

თვისება	ტიპი	აღწერა
BlockSize	Int	დასაშიფრი ან გასაშიფრი ბაიტების რაოდენობა ერთი ოპერაციის შესრულებისას
FeedbackSize	Int	მონაცემების რაოდენობა თითოეული ბლოკიდან, რომლებიც გამოიყენება მომდევნო ბლოკის დასაშიფრად
IV	Byte	მაინიციალიზებელი ვექტორი მოცემული ალგორითმისთვის
Key	Byte	დაშიფვრის გასაღები მოცემული ალგორითმისთვის
KeySize	Int	გასაღებში ბიტების რაოდენობა
LegalBlockSizes	array<KeySizes^>^	ბლოკების ზომები, რომლებსაც მოცემული ალგორითმი უზრუნველყოფს
LegalKeySizes	array<KeySizes^>^	გასაღებების ზომები, რომლებსაც მოცემული ალგორითმი უზრუნველყოფს
Mode	CipherMode	განსაზღვრავს სიმეტრიული ალგორითმის ოპერაციის რეჟიმს
Padding	PaddingMode	განსაზღვრავს ბაიტებს, რომლებიც დაემატება უკანასკნელ ბლოკს, რომელსაც არ აქვს საკმარისი სიგრძე

ცხრილი 15.13. TripleDESCryptoServiceProvider კლასის ფუნქციები

ფუნქცია	აღწერა
void Clear()	ათავისუფლებს ალგორითმის მიერ დაკავებულ რესურსებს
ICryptoTransform^ CreateDecryptor()	ქმნის ობიექტს, რომლის გამოყენება შეიძლება გაშიფვრის ოპერაციის შესასრულებლად
ICryptoTransform^ CreateEncryptor()	ქმნის ობიექტს, რომლის გამოყენება შეიძლება დაშიფვრის ოპერაციის შესასრულებლად
void GenerateIV()	ახდენს ახალი შემთხვევითი მაინიციალიზებელი ვექტორის გენერირებას
void GenerateKey()	ახდენს ახალი შემთხვევითი გასაღების გენერირებას
bool ValidateKeySize(int *სიგრძე*)	განსაზღვრავს, აქვს თუ არა მიმდინარე გასაღებს კორექტული ზომა

პროგრამაში ჯერ იქმნება TripleDESCryptoServiceProvider კლასის ობიექტი მონაცემების დაშიფრავი სამსახურის უზრუნველყოფისათვის. შემდეგ იქმნება CryptoStream კლასის

ობიექტი, რომელიც დაშიფრავს მონაცემებს. CryptoStream კლასის კონსტრუქტორს სამი პარამეტრი გადაეცემა. პირველია ობიექტი-ნაკადი, რომელიც გამოიყენება ფაილში მონაცემების ჩასაწერად ან წასაკითხად. მეორე პარამეტრია ფუნქცია, რომელიც დაშიფრავს ან გაშიფრავს მონაცემებს. მესამე პარამეტრია რეჟიმი, რომელიც განსაზღვრავს CryptoStream ობიექტი კითხულობს თუ წერს მონაცემებს.

პროგრამაში StreamWriter ობიექტიდან მონაცემები გადაეცემა CryptoStream ობიექტს, რომელიც დაშიფრავს ამ მონაცემებს, შემდეგ კი დაშიფრული მონაცემები გადაეცემა FileStream ობიექტს, რომელიც ამ მონაცემებს ფაილში ჩაწერს. ბოლოს პროგრამა ფაილში წერს გასაღებსა და მაინიაციალიზებელ ვექტორს.

მოცემული პროგრამით ხდება ფაილიდან წაკითხული მონაცემების გაშიფვრა.

```
{
//      ფაილიდან წაკითხული მონაცემების გაშიფვრა
TripleDESCryptoServiceProvider^ CriptoProvaideri = gcnew TripleDESCryptoServiceProvider();
FileStream^   SafailoNakadi1 = File::OpenRead("encrypted.key");
BinaryReader^ OrobitiWamkitxavi = gcnew BinaryReader(SafailoNakadi1);
//      ფაილიდან გასაღებისა და ვექტორის წაკითხვა
CriptoProvaideri->Key = OrobitiWamkitxavi->ReadBytes(24);
CriptoProvaideri->IV  = OrobitiWamkitxavi->ReadBytes(8);
FileStream^   SafailoNakadi2 = File::OpenRead("temp.txt");
//      ფაილიდან წაკითხული მონაცემების გაშიფვრა
CryptoStream^ CriptoNakadi =
gcnew CryptoStream(SafailoNakadi2, CriptoProvaideri->CreateDecryptor(), CryptoStreamMode::Read);
StreamReader^ WakitxvisNakadi = gcnew StreamReader(CriptoNakadi);
label1->Text = WakitxvisNakadi->ReadToEnd()->ToString() + "\n" + L"გასაღების ზომა = " +
                CriptoProvaideri->KeySize.ToString() + "\n" + L"ბლოკის ზომა = " +
                CriptoProvaideri->BlockSize.ToString();
label2->Text = L"რეჟიმი = " + CriptoProvaideri->Mode.ToString();
WakitxvisNakadi->Close();
}
```

პროგრამაში ხდება დაშიფრული მონაცემების წაკითხვა FileStream ობიექტის მიერ, მათი გაშიფვრა CryptoStream ობიექტის მიერ, შემდეგ კი მათი გადაცემა StreamReader ობიექტისათვის და ეკრანზე გამოტანა.

ასიმეტრიული კრიპტოგრაფია

ასიმეტრიული კრიპტოგრაფიის ალგორითმები განსაზღვრულია არა ნაკადებთან, არამედ, მცირე ზომის მონაცემებთან სამუშაოდ. მოცემული პროგრამით ხდება მონაცემების დაშიფვრისა და გაშიფვრის დემონსტრირება ასიმეტრიული კრიპტოგრაფიის ფუნქციების გამოყენებით.

```
{
//      მონაცემების დაშიფვრისა და გაშიფვრის დემონსტრირება
//      ასიმეტრიული კრიპტოგრაფიის ფუნქციების გამოყენებით
label1->Text = "";
//      ახალი კრიპტოგრაფიული პროვაიდერის შექმნა
RSACryptoServiceProvider^ CriptoProvaideri = gcnew RSACryptoServiceProvider();
array<Byte>^ DasashifriMasivi = gcnew array<Byte> { 1, 2, 3, 4, 5 };
//      მონაცემების დაშიფვრა
```

```
array<Byte>^ DashifruliMasivi = CriptoProvaideri->Encrypt(DasashifriMasivi, false);
label1->Text = L"დაშიფრული მონაცემები: \n";
for ( int indexi = 0; indexi < DashifruliMasivi->GetLength(0); indexi++ )
label1->Text += DashifruliMasivi[indexi].ToString() + "   ";
label1->Text += "\n";
//        მონაცემების გაშიფვრა
array<Byte>^ GashifruliMasivi = CriptoProvaideri->Decrypt(DashifruliMasivi, false);
label1->Text += L"გაშიფრული მონაცემები: \n";
for ( int indexi = 0; indexi < GashifruliMasivi->GetLength(0); indexi++ )
label1->Text += GashifruliMasivi[indexi].ToString() + "   ";
label2->Text += L"ალგორითმის სახელი - " +
                        CriptoProvaideri->KeyExchangeAlgorithm->ToString() + "\n" +
                        L"გასაღების ზომა = "  + CriptoProvaideri->KeySize.ToString() + "\n";
}
```

სერიულ პორტთან მუშაობა

მკითხველმა უნდა იცოდეს სერიული პორტის (COM პორტის) მუშაობის პრინციპები, რადგან წიგნის მიზანი არ არის მათი დეტალური განხილვა. სერიულ პორტთან მუშაობას პროგრამისტის კუთხით განვიხილავთ. პორტთან სამუშაოდ SerialPort კლასი გამოიყენება. მისი ზოგიერთი თვისება და ფუნქცია მოცემულია 15.14-15.15 ცხრილებში.

სერიულ პორტთან მუშაობა კონკრეტული მაგალითით განვიხილოთ. დავუშვათ, კომპიუტერში გვინდა მივიღოთ მიკროპროცესორული მოწყობილობის ორი რეგისტრის შემცველობა. მიკროპროცესორული მოწყობილობა მუშაობს MODBUS პროტოკოლით და კომპიუტერს COM პორტის საშუალებით უკავშირდება. აქვე შევნიშნოთ, რომ პროტოკოლის არჩევა კონკრეტულ მოწყობილობაზეა დამოკიდებული. მოწყობილობის თითოეული რეგისტრი ოთხბაიტიანია. ამ პროტოკოლის მიხედვით კომპიუტერმა მოწყობილობას უნდა გაუგზავნოს ინფორმაცია, რომელსაც შემდეგი ფორმატი აქვს:

პირველი ბაიტი - მოწყობილობის მისამართი;
მეორე ბაიტი - წაკითხვის ბრძანება;
მესამე და მეოთხე ბაიტები - მოწყობილობის იმ რეგისტრის მისამართი, საიდანაც ინფორმაცია უნდა მივიღოთ;
მეხუთე და მეექვსე ბაიტები - რეგისტრების რაოდენობა;
მეშვიდე და მერვე ბაიტები - საკონტროლო ჯამი.

ამ ბრძანების საპასუხოდ მოწყობილობა დაახლოებით 600 მილიწამის შემდეგ კომპიუტერს უგზავნის ინფორმაციას, რომელსაც პროტოკოლის მიხედვით, შემდეგი ფორმატი აქვს:

პირველი ბაიტი - მოწყობილობის მისამართი;
მეორე ბაიტი - წაკითხვის ბრძანება;
მესამე ბაიტი - ბაიტების რაოდენობა;
მეოთხე, მეხუთე, მეექვსე და მეშვიდე ბაიტები - პირველი რეგისტრის მნიშვნელობები;
მერვე, მეცხრე, მეათე და მეთერთმეტე ბაიტები - მეორე რეგისტრის მნიშვნელობები;
მეთორმეტე და მეცამეტე ბაიტები - საკონტროლო ჯამი.

ოთხბაიტიანი რეგისტრიდან მიღებული მონაცემი მთელ რიცხვად შეგვიძლია შემდეგნაირად გარდავქმნათ:

$Y = X_1 * 256^3 + X_2 * 256^2 + X_3 * 256^1 + X_4 * 356^0$

აქ Y გარდაქმნის შედეგად მიღებული მთელი რიცხვია, X_1 რეგისტრის უფროსი ბაიტია, X_2 - რეგისტრის მომდევნო ბაიტი და ა.შ. გარდაქმნის შედეგად მიღებული მთელი რიცხვები შეგვიძლია ჩავწეროთ ფაილში ან მონაცემთა ბაზაში (შემდგომი დამუშავების მიზნით), ან ავსახოთ გრაფიკის სახით.

კონკრეტულად თუ რომელი პორტი გამოყო ოპერაციულმა სისტემამ მიკროპროცესორულ მოწყობილობასთან სამუშაოდ, შეგვიძლია ვნახოთ Device Manager ფანჯარაში. სწორედ ამ გამოყოფილი პორტის სახელი უნდა მივუთითოთ პროგრამაში პორტის პარამეტრების განსაზღვრისას.

ზემოთ თქმულის დემონსტრირება ხდება მოცემული პროგრამით. პროგრამის შეტანამდე using namespace დირექტივების ბლოკს უნდა დავუმატოთ using namespace System::IO::Ports; დირექტივა.

```
{
//      სერიულ პორტთან მუშაობის დემონსტრირება
BinaryWriter^ file_out= gcnew BinaryWriter(gcnew FileStream("faili.dat", FileMode::Create));
long long ricxvi1, ricxvi2;
int wakitxuli_baitebis_raodenoba;
//      ბუფერი მონაცემების მისაღებად
array<Byte>^ buf13 = gcnew array<Byte>(13);
//      ბუფერი მონაცემების გასაგზავნად
array<Byte>^ buf8 = gcnew array<Byte>(8);
SerialPort^ COM_Porti_1;
//      მოწყობილობისთვის გასაგზავნი მონაცემების მომზადება
buf8[0] = 2;                                    //      მოწყობილობის მისამართი
buf8[1] = 3;                                    //      მონაცემების წაკითხვის ბრძანება
buf8[2] = 29;                                   //      რეგისტრის მისამართი
buf8[3] = 79;
buf8[4] = 0;                                    //      რეგისტრების რაოდენობა
buf8[5] = 2;
buf8[6] = 243;                                  //      საკონტროლო ჯამი
buf8[7] = 131;
//      პორტის პარამეტრების მომზადება
COM_Porti_1->PortName = textBox1->Text;  //      პორტის სახელი
COM_Porti_1->BaudRate = 38400;          //      პორტის სწრაფქმედება
COM_Porti_1->Parity = Parity::None;     //      პარიტეტი არ გვაქვს
COM_Porti_1->DataBits = 8;              //      ბაიტში მონაცემების ბიტების რაოდენობა
COM_Porti_1->Open();                    //      პორტის გახსნა
//      პორტში მონაცემების გაგზავნა
COM_Porti_1->Write(buf8,0,8);
Thread::Sleep(600);                             //      დაყოვნება 600 მილიწამი
//      პორტიდან მონაცემების მიღება
wakitxuli_baitebis_raodenoba = COM_Porti_1->Read(buf13,0,13);
//      რეგისტრებიდან მიღებული მონაცემების გარდაქმნა
ricxvi1 = buf13[3] * 256 * 256 * 256 + buf13[4] * 256 * 256 + buf13[5] * 256 + buf13[6];
ricxvi2 = buf13[7] * 256 * 256 * 256 + buf13[8] * 256 * 256 + buf13[9] * 256 + buf13[10];
//      მონაცემების ჩაწერა ფაილში
file_out->Write(ricxvi1);
file_out->Write(ricxvi2);
```

```
file_out->Close();
//
label1->Text = COM_Porti_1->PortName;                    //        პორტის სახელის ეკრანზე გამოტანა
label2->Text = wakitxuli_baitebis_raodenoba.ToString();
label3->Text =  L"წაკითხული ბაიტების რაოდენობა = " + COM_Porti_1->BytesToRead.ToString() +
                L"\nჩაწერილი ბაიტების რაოდენობა = " + COM_Porti_1->BytesToWrite.ToString();
label4->Text = ricxvi1.ToString();
label5->Text = ricxvi2.ToString();
//
COM_Porti_1->Close();                                    //        პორტის დახურვა
}
```

ცხრილი 15.14. SerialPort კლასის თვისებები

თვისება	აღწერა
int BaudRate	სერიული პორტის მუშაობის სიხშირე
int BytesToRead	განსაზღვრავს მონაცემების ბაიტების რაოდენობას მიმღებ ბუფერში
int BytesToWrite	განსაზღვრავს მონაცემების ბაიტების რაოდენობას გადაცემის ბუფერში
int DataBits	განსაზღვრავს მონაცემების ბიტების სტანდარტულ რაოდენობას 1 ბაიტში. მისი მნიშვნელობაა 5÷8
Encoding: Encoding	განსაზღვრავს ბაიტის კოდირებას ტექსტის გადაცემამდე ან გადაცემის შემდეგ გარდაქმნისთვის
Handshake Handshake	განსაზღვრავს პროტოკოლს სერიული პორტით მონაცემების გადაცემისათვის. იღებს None, XonXoff, RequestToSend, RequestToSendXonXoff მნიშვნელობებს
bool IsOpen	შეიცავს true-ს თუ სერიული პორტი ღიაა და false-ს თუ პორტი დახურულია
String^ NewLine	განსაზღვრავს სიდიდეს, რომელიც გამოიყენება გამომახების დასასრულის ინტერპრეტირებისათვის ReadLine() და WriteLine() ფუნქციებში
Parity Parity	განსაზღვრავს პარიტეტის შემოწმების პროტოკოლს. იღებს None, Odd, Even, Mark, Space მნიშვნელობებს
String^ PortName	განსაზღვრავს სერიული პორტის სახელს
int ReadBufferSize	განსაზღვრავს სერიული პორტის შესასვლელი (მიმღები) ბუფერის ზომას
int ReadTimeout	განსაზღვრავს მილიწამების რაოდენობას, რომელიც უნდა გავიდეს მანამ, სანამ ტაიმ-აუტი დადგებოდეს, როდესაც კითხვის ოპერაცია არ არის დამთავრებული
StopBits StopBits	განსაზღვრავს სტოპ-ბიტების რაოდენობას. ის იღებს None, One, Two, OnePointFive მნიშვნელობებს
int WriteBufferSize	განსაზღვრავს სერიული პორტის გამოსასვლელი (გადაცემის) ბუფერის ზომას
int WriteTimeout	განსაზღვრავს მილიწამების რაოდენობას, რომელიც უნდა გავიდეს მანამ, სანამ ტაიმ-აუტი დადგებოდეს, როდესაც ჩაწერის ოპერაცია არ არის დამთავრებული

ცხრილი 15.15. SerialPort კლასის ფუნქციები

ფუნქცია	აღწერა
void Close()	ხურავს სერიულ პორტთან შეერთებას, IsOpen თვისებას false მნიშვნელობას ანიჭებს და ათავისუფლებს შესაბამის რესურსებს
void Dispose()	ათავისუფლებს პორტის მიერ დაკავებულ უმართავ რესურსებს
array<String^>^ GetPortNames()	გასცემს კომპიუტერის სერიული პორტების სახელებს
void Open()	ხსნის ახალ შეერთებას სერიულ პორტთან
int Read(array<Byte>^ ბუფერი, int წანაცვლება, int რაოდენობა)	სერიული პორტიდან კითხულობს მითითებული რაოდენობის ბაიტებს და ათავსებს მათ ბუფერში დაწყებული *წანაცვლებით* განსაზღვრული პოზიციიდან
int Read(array<Char>^ ბუფერი, int წანაცვლება, int რაოდენობა)	სერიული პორტიდან კითხულობს მითითებული რაოდენობის სიმბოლოებს და ათავსებს მათ ბუფერში დაწყებული *წანაცვლებით* განსაზღვრული პოზიციიდან
int ReadByte()	სინქრონულად კითხულობს ერთ ბაიტს შესასვლელი ბუფერიდან
int ReadChar()	სინქრონულად კითხულობს ერთ სიმბოლოს შესასვლელი ბუფერიდან
String^ ReadExisting()	შესასვლელ ნაკადში ათავსებს კოდირებაზე დაფუძნებულ ყველა უშუალოდ მისაწვდომ ბაიტს
String^ ReadLine()	მონაცემებს კითხულობს შესასვლელი ბუფერიდან NewLine სიმბოლომდე
String^ ReadTo(String^ *სტრიქონი*)	კითხულობს სტრიქონს მითითებულ სიმბოლომდე
void Write(String^ *სტრიქონი*)	მითითებულ სტრიქონს უგზავნის სერიულ პორტს
void Write(array<Byte>^ *ბუფერი*, int *წანაცვლება*, int *რაოდენობა*)	ბუფერიდან სერიულ პორტს უგზავნის მითითებული რაოდენობის ბაიტს
void Write(array<Char>^ *ბუფერი*, int *წანაცვლება*, int *რაოდენობა*)	ბუფერიდან სერიულ პორტს უგზავნის მითითებული რაოდენობის სიმბოლოს
void WriteLine(String^ *სტრიქონი*)	მითითებულ სტრიქონს და NewLine სიმბოლოს ათავსებს გამოსასვლელ ბუფერში

asm საკვანძო სიტყვა

C++ ენაზე პროგრამის შედგენისას მისი ფრაგმენტი შეგვიძლია ასემბლერ ენაზე დავწეროთ. ამისთვის გამოიყენება asm საკვანძო სიტყვა. მისი სინტაქსია:

__asm ***ასემბლერის_ბრძანება*** **[;]**

__asm { *ასემბლერის_ბრძანების_სია* } [;]

იმისათვის, რომ C++ ენის კოდში მოვათავსოთ ასემბლერ ენაზე დაწერილი პროგრამა, უნდა შევქმნათ არაუსაფრთხო კოდი. ამისათვის, პროექტის შექმნისას, Template ფანჯარაში (ნახ. 2.2) უნდა ავირჩიოთ Win32 Console Application ელემენტი. გახსნილ ფანჯარაში (ნახ. 15.1) ვაჭერთ Finish კლავიშს და შეგვაქვს პროგრამა ისე, როგორც ქვემოთ არის ნაჩვენები:

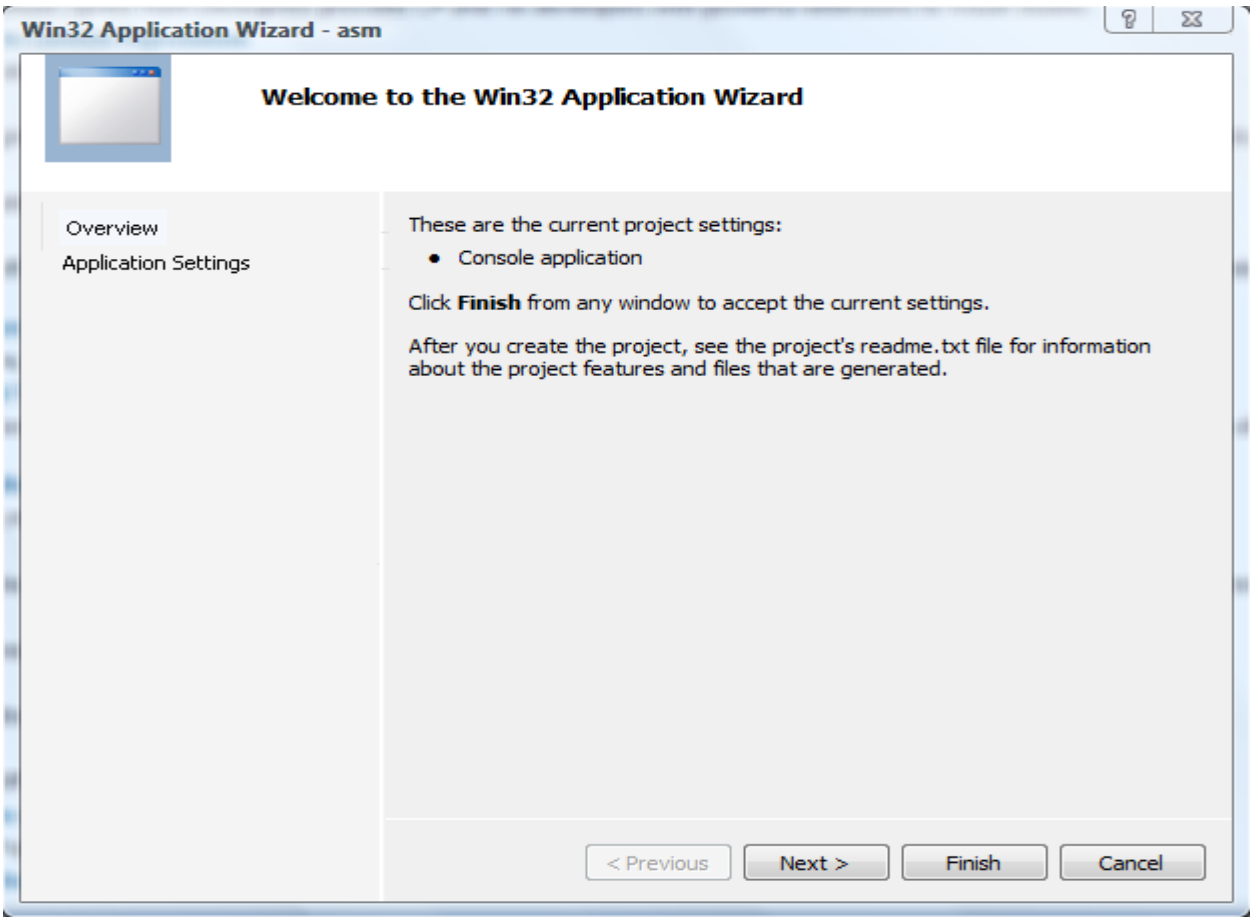
```
//
#include "stdafx.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[ ])
{
short ricxvi = 5;
//        ასემბლერ ენაზე შედგენილი კოდი
__asm
        {
          mov ax, ricxvi
          mov bx, 10
          add ax, bx
          mov ricxvi, ax
        }
cout << "SUM = " << ricxvi;              //        შედეგის გამოტანა ეკრანზე
cin >> ricxvi;                          //        ეკრანზე შედეგის დაყოვნება რიცხვის შეტანამდე
return 0;
}
        C++ პროგრამაში შეგვიძლია ჩავრთოთ ასემბლერ ენის ერთი ბრძანება. მაგალითი.
//
//
#include "stdafx.h"
#include <iostream>
#include <fstream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[ ])
{
short ricxvi;
//        ასემბლერ ენის ბრძანება
__asm  mov ricxvi, 55
cout << ricxvi;
cin >> ricxvi;
return 0;
}
```


Win32 Application Wizard - asm
Welcome to the Win32 Application Wizard
Overview
Application Settings
These are the current project settings:
Console application
Click Finish from any window to accept the current settings.
After you create the project, see the project's readme.txt file for information about the project features and files that are generated.
< Previous
Next >
Finish
Cancel

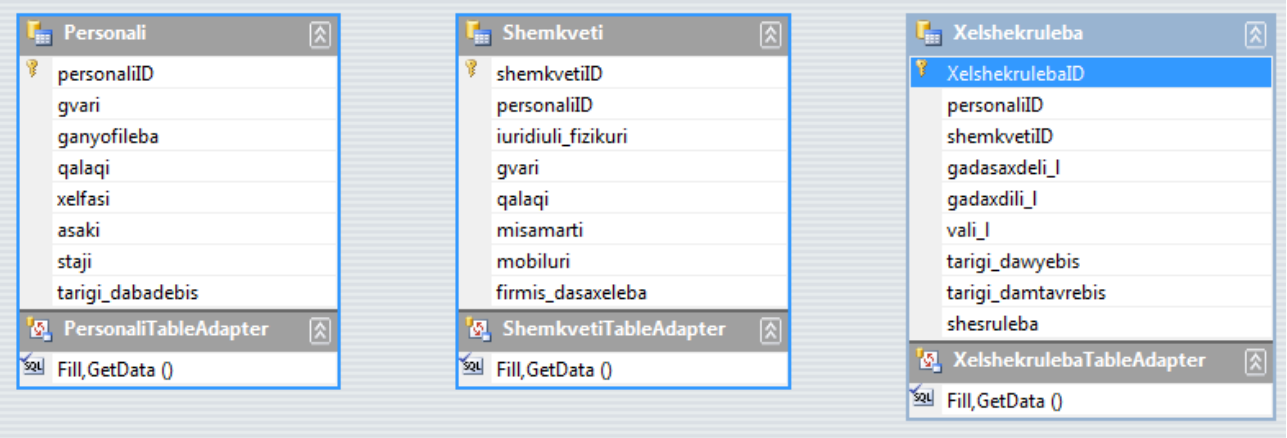
ნახ. 15.1.

თავი 17. ADO.NET კლასები

მონაცემთა ბაზასთან მიმართვა Visual Studio .NET-ის საშუალებებით

ამ თავში ვნახავთ თუ როგორ უნდა ვიმუშაოთ SQL სერვერზე მოთავსებულ მონაცემთა ბაზასთან C++ პროგრამიდან Active Data Objects ბიბლიოთეკის კლასების გამოყენებით .NET Framework პლატფორმისათვის (შემოკლებით ADO.NET). აქვე შევნიშნავ, რომ მკითხველს უნდა ჰქონდეს მინიმალური ცოდნა SQL სერვერის შესახებ. ქართულ ენაზე SQL სერვერი აღწერილია, იხ. [2].

დავუშვათ, სერვერზე შევქმენით Shekveta მონაცემთა ბაზა, რომელიც შემდეგ ობიექტებს შეიცავს: Personali, Shemkveti და Xelshekruleba ცხრილებს, View_3 წარმოდგენას, Chemi_Proc და Myproc_Par შენახულ პროცედურებს და Maqsimaluri_Xelfasi ფუნქციას. ცხრილების სტრუქტურა ასეთია:



თითოეულ ცხრილს უნდა ჰქონდეს პირველადი გასაღები. ეს ცხრილები შევავსოთ მონაცემებით. Chemi_Proc შენახული პროცედურა გასცემს Personali ცხრილის სტრიქონებს. Myproc_Par შენახულ პროცედურას გამოაქვს ინფორმაცია მითითებული განყოფილების თანამშრომლების შესახებ. Maqsimaluri_Xelfasi ფუნქცია გამოითვლის და გასცემს მითითებული განყოფილების თანამშრომლების მაქსიმალურ ხელფასს.

SQL სერვერზე მოთავსებულ მონაცემთა ბაზასთან მიმართვა ADO.NET ბიბლიოთეკის კლასების გამოყენებით შემდეგნაირად ხორციელდება: C++ პროგრამა უერთდება სერვერზე მოთავსებულ მონაცემთა ბაზას, იღებს საჭირო ობიექტებს, როგორიცაა ცხრილები, წარმოდგენები, ფუნქციები და შენახული პროცედურები, და მათ იმ კომპიუტერის (კლიენტის) მეხსიერებაში ათავსებს, რომელზეც ეს პროგრამა მუშაობს. ამ ობიექტების სიმრავლეს მონაცემების ლოკალური ნაკრები ეწოდება. ამის შემდეგ, ჩვენ მოგვეცემა ამ ობიექტებთან მუშაობის შესაძლებლობა. კერძოდ, შევძლებთ ცხრილების სტრიქონების ცვლილებას, ფუნქციებისა და შენახული პროცედურების გამოძახებას და ა.შ. C++ პროგრამა პერიოდულად უნდა დავუკავშიროთ მონაცემთა ბაზას იმ ცვლილებების შესანახად, რომელიც ჩვენ შევიტანეთ მონაცემების ლოკალურ ნაკრებში (ასლში).

მონაცემთა ბაზა შეიძლება იყოს იმ კომპიუტერზე, რომელზეც ჩვენი პროგრამა მუშაობს. თუმცა, ყველაფერი დამოკიდებულია გადასაწყვეტი ამოცანის მოთხოვნებზე. უმჯობესია მონაცემთა ბაზის მოთავსება უფრო მძლავრ კომპიუტერზე (სერვერზე), რომელიც ერთდროულად მოემსახურება იმ პროგრამებსა და მომხმარებლებს, რომლებიც SQL მოთხოვნებს აგზავნიან.

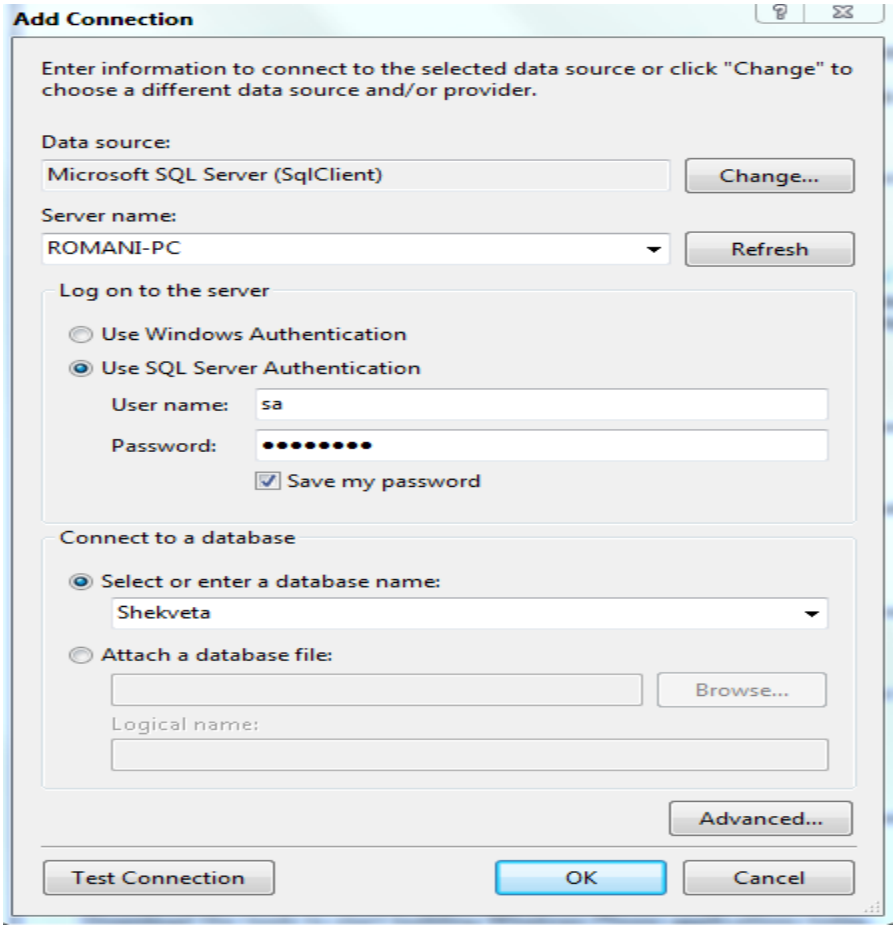
Visual Studio .NET გარემოს აქვს ფუნქციები, რომლებიც საშუალებას იძლევა დავუკავშირდეთ მონაცემთა ბაზას და ვიმუშაოთ მის ცხრილებთან. ეს ფუნქციები

თავმოყრილია Server Explorer ფანჯარაში.

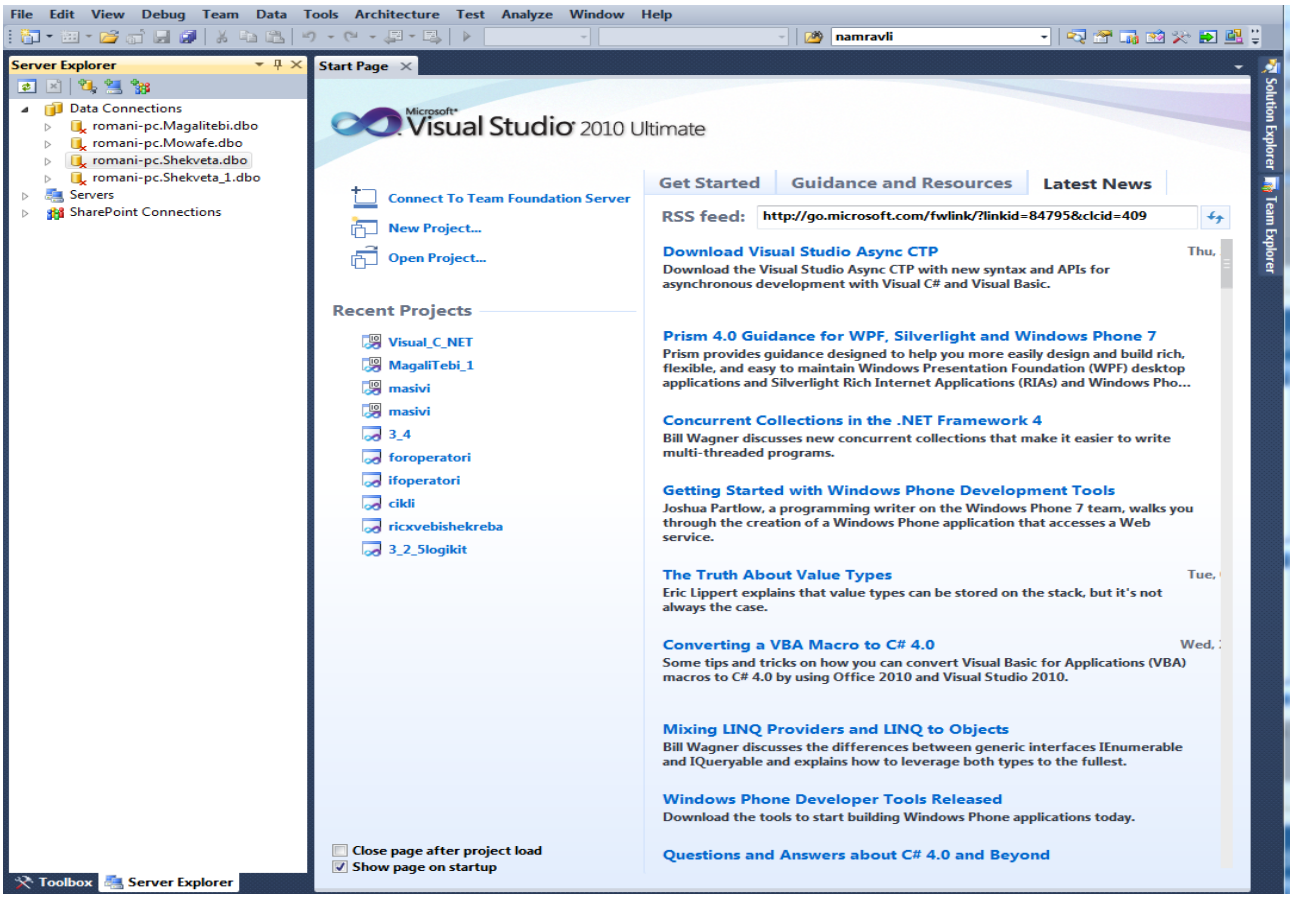
უწინარეს ყოვლისა, მონაცემთა ბაზასთან უნდა დავამყაროთ კავშირი (შეერთება). ამისათვის, ჯერ უნდა გავუშვათ Visual Studio .NET სისტემა, შემდეგ კი შევასრულოთ Tools მენიუს Connect to Database ბრძანება. გაიხსნება Add Connection ფანჯარა (ნახ. 16.1). Server name ველში შეგვაქვს სერვერის (კომპიუტერის) სახელი. შემდეგ უნდა ჩავრთოთ Use SQL Server Authentication გადამრთველი და User name და Password ველებში შევიტანოთ შესაბამისი ინფორმაცია. საჭირო ბაზას ვირჩევთ Select or enter a database name გაშლადი სიიდან. ვაჭერთ Ok კლავიშს.

მონაცემთა ბაზასთან კავშირის დამყარების შემდეგ, შევძლებთ ცხრილების ნახვას და ცვლილების შეტანას. ცხრილებთან მიმართვისათვის ვხსნით Server Explorer ფანჯარას (ნახ. 16.2), რისთვისაც ერთხელ ვაჭერთ გახსნილი ფანჯრის მარცხნივ მოთავსებულ Server Explorer პანელის პიქტოგრამას. თუ ეს პანელი არ ჩანს, მაშინ უნდა შევასრულოთ View მენიუს Server Explorer ბრძანება. ამ ფანჯრის კატალოგების ხის უბანში ჯერ ვხსნით ROMANI-PC.Shekveta.dbo განშტოებას, შემდეგ კი - Tables განშტოებას (ნახ. 16.3). Personali ცხრილიდან სტრიქონების მისაღებად ისარს მივიტანთ ამ ცხრილის პიქტოგრამასთან, ვხსნით კონტექსტურ მენიუს და ვასრულებთ Show Table Data ბრძანებას. 16.4 ნახაზზე ნაჩვენებია Personali ცხრილის სტრიქონები.

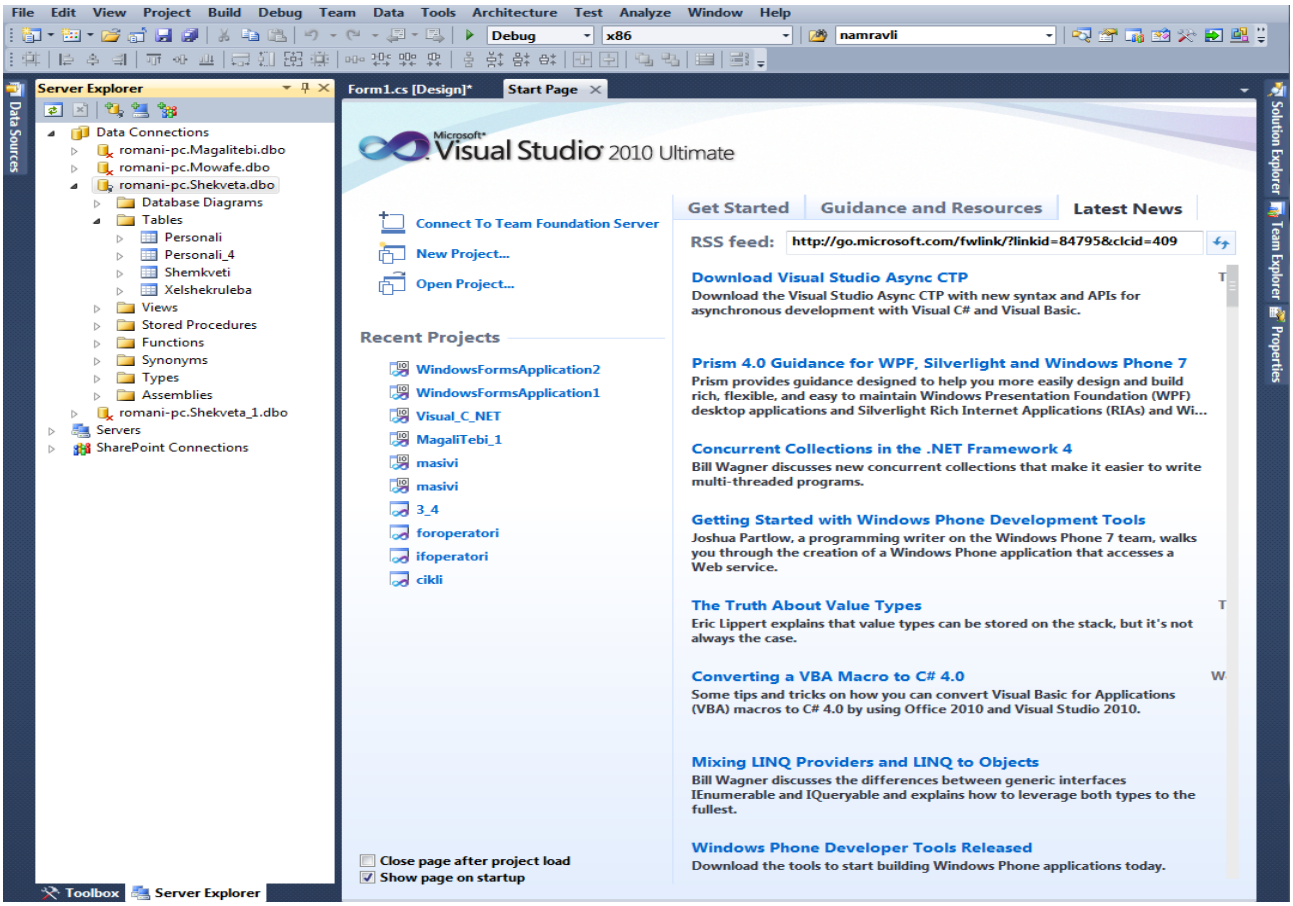
SQL მოთხოვნის შესატანად ვაჭერთ ინსტრუმენტების პანელის Show SQL Pane კლავიშს (ნახ. 16.5). მოთხოვნის შესასრულებლად ვაჭერთ Ctrl+R კლავიშებს ან ინსტრუმენტების პანელის  კლავიშს. SQL მოთხოვნა შეგვიძლია, აგრეთვე, ავაგოთ ვიზუალურად თუ დავაჭერთ Show Diagram Pane კლავიშს (ნახ. 16.6). შეგვიძლია სვეტების თვისებების ნახვა თუ გავხსნით Personal განშტოებას, მოვნიშნავთ საჭირო სვეტს, გავხსნით კონტექსტურ მენიუს და შევასრულებთ Properties ბრძანებას.



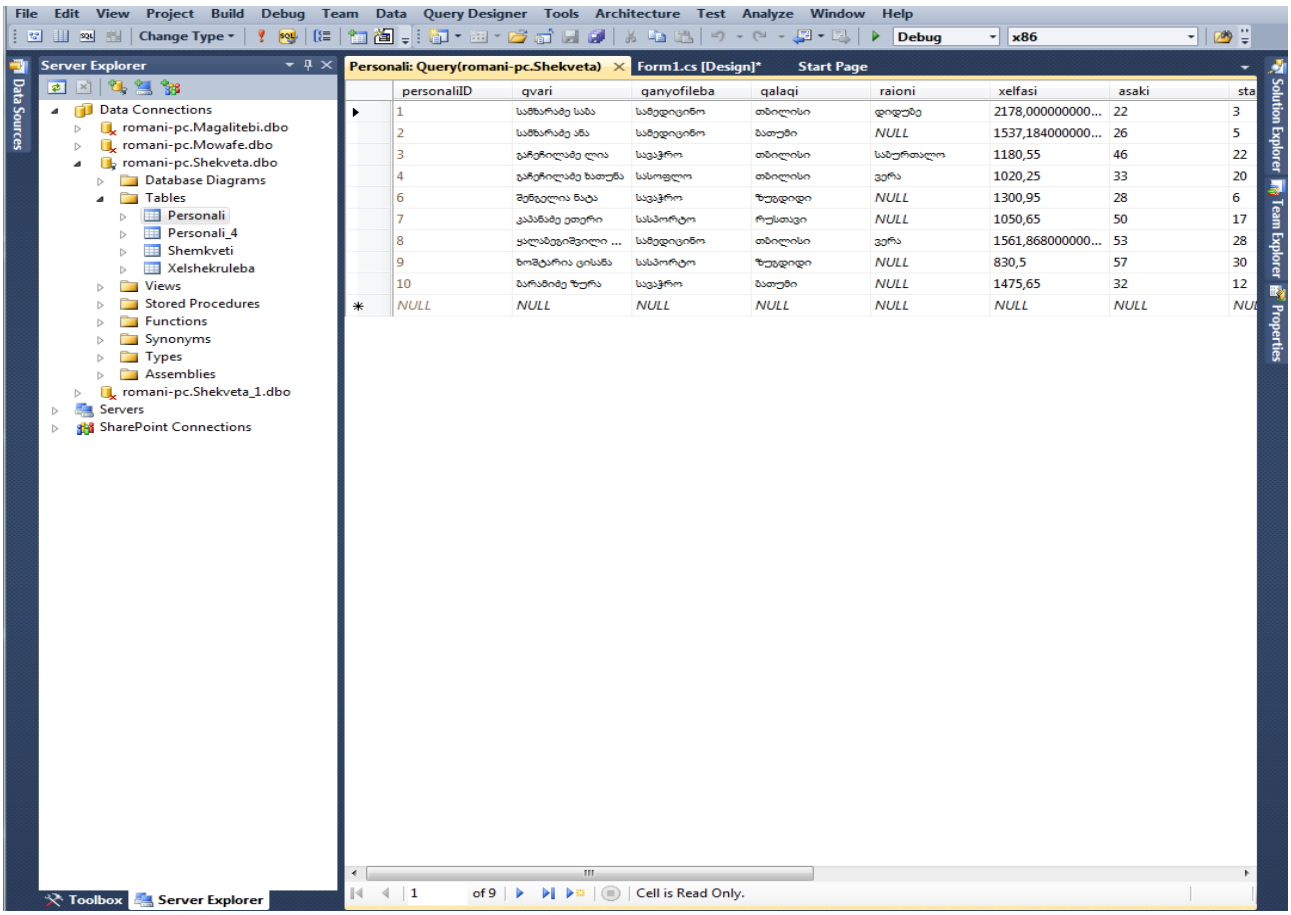
ნახ. 16.1.



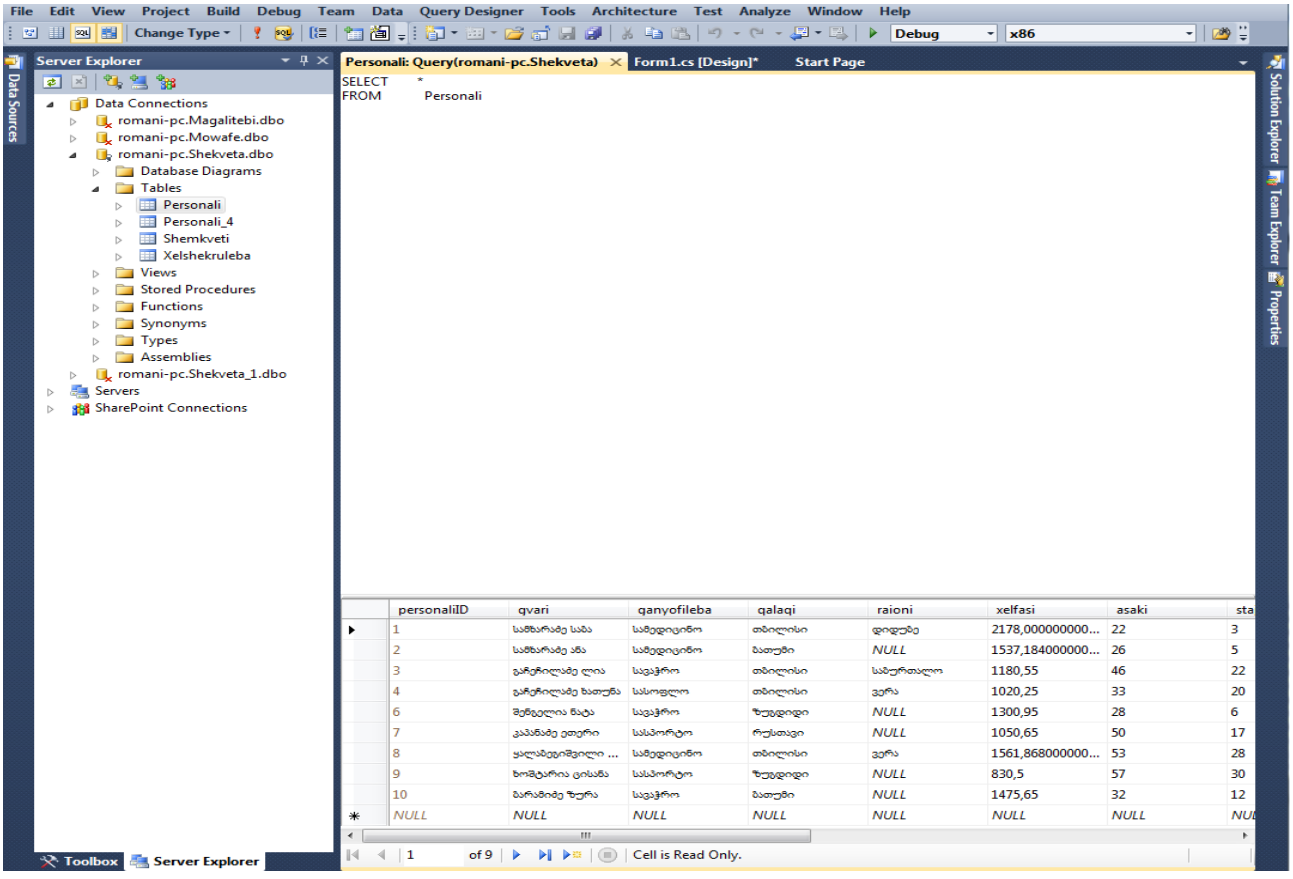
ნახ. 16.2.



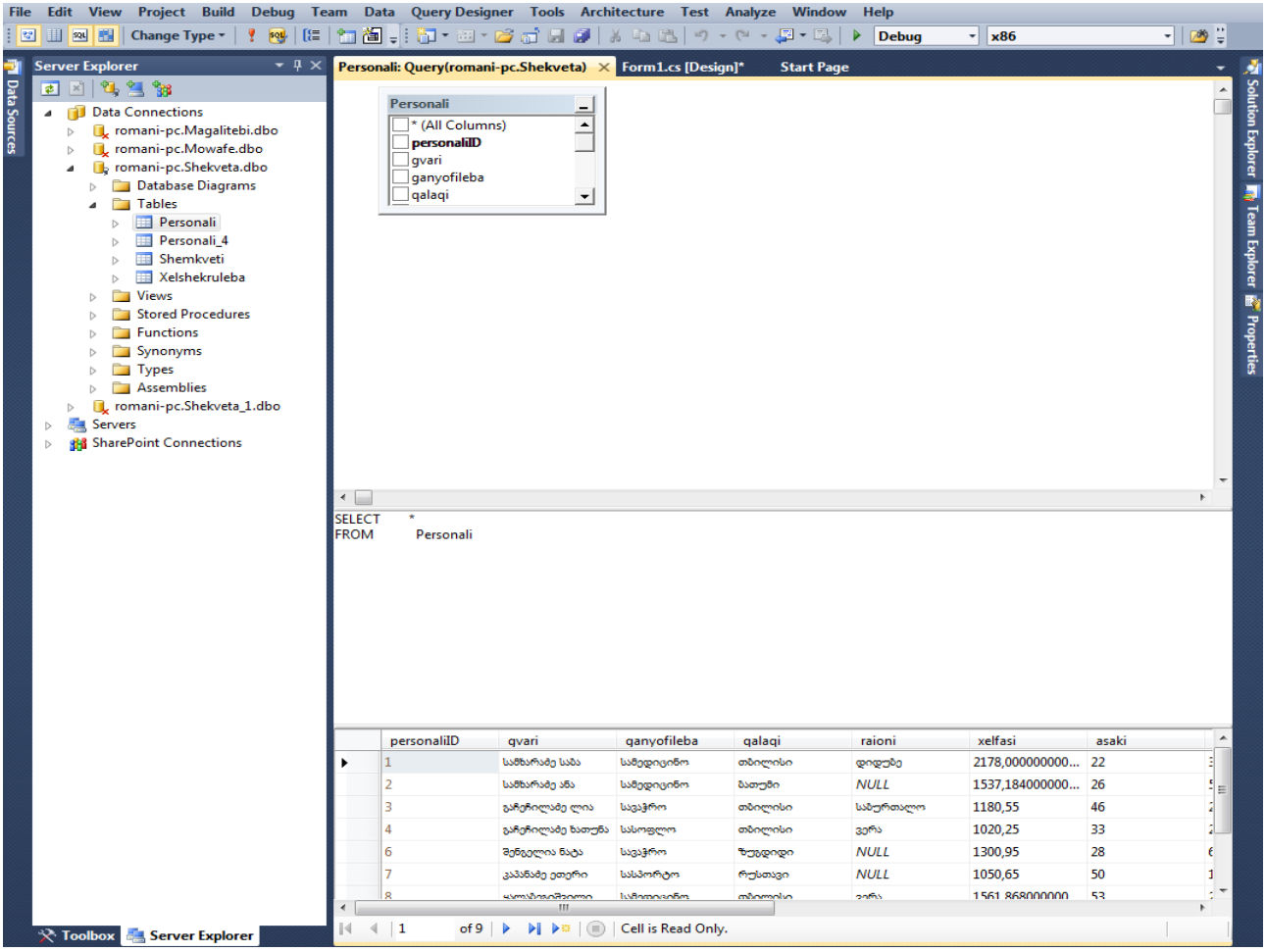
ნახ. 16.3.



ნახ. 16.4.



ნახ. 16.5.



ნახ. 16.6.

ADO.NET კლასების მომიხილვა

ADO.NET არის კლასების სიმრავლე, რომელიც გამოიყენება C++ და .NET Framework გარემოსთან ერთად სხვადასხვა ფორმატის მონაცემთა ბაზასთან სამუშაოდ. ADO.NET ტექნოლოგია ინტეგრირებულია .NET Framework გარემოსთან და ორიენტირებულია .NET-ის ნებისმიერ ენასთან, განსაკუთრებით კი – C++-თან სამუშაოდ. ის მოიცავს System::Data სახელების სივრცეს და მასში შემავალ ჩადგმულ სახელების სივრცეებს: System::Data::SqlClient და ა.შ.

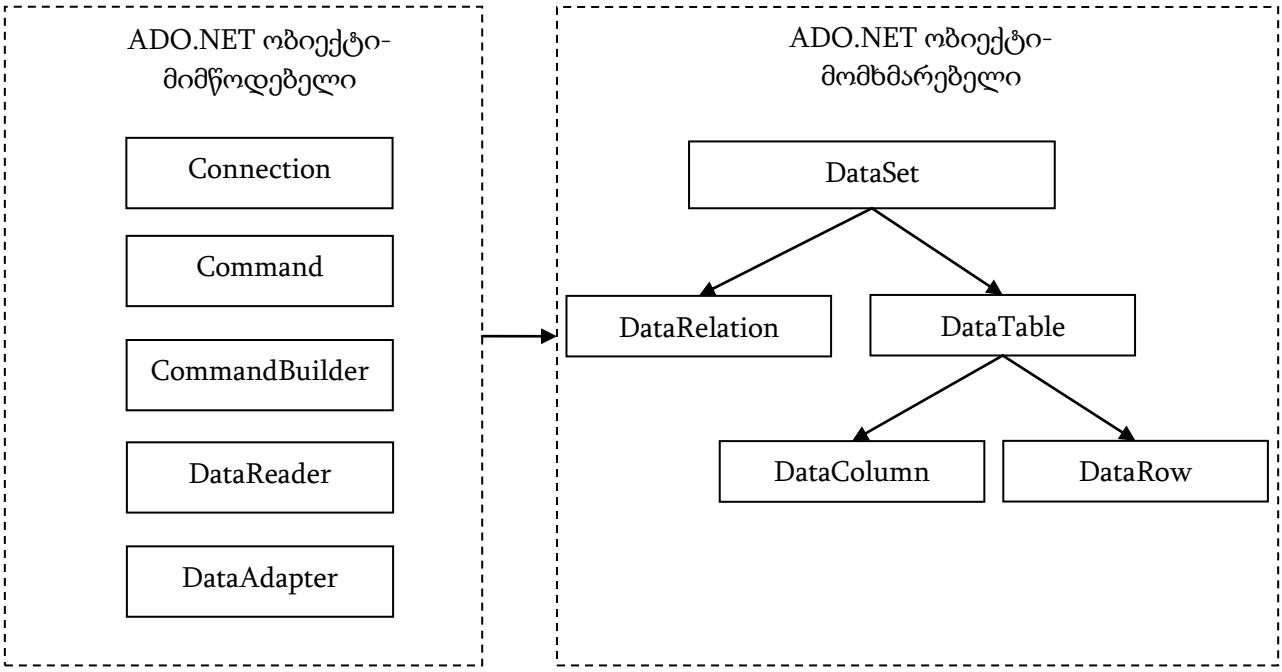
ADO.NET უზრუნველყოფს მონაცემთა რელაციურ ბაზებთან: როგორიცა Microsoft SQL Server და Microsoft Access, მარტივ მიმართვას. მისი კლასების საშუალებით შეგვიძლია განვსაზღვროთ ცხრილები, სვეტები და სტრიქონები. მასში განსაზღვრულია, აგრეთვე DataSet კლასი, რომელიც რელაციური ცხრილების ერთობლიობაა, მათ შორის არსებულ კავშირებთან ერთად.

სხვადასხვა ფორმატის მონაცემთა ბაზებთან მუშაობის გასამარტივებლად 90-იანი წლების დასაწყისში Microsoft-მა შეიმუშავა ODBC (Open Database Connectivity – მონაცემთა ბაზებთან ურთიერთქმედების ღია ინტერფეისი) ინტერფეისი. მას აქვს ფუნქციების სრული ნაკრები, რომლების გამოყენებაც შეიძლება სხვადასხვა ფორმატის მონაცემთა ბაზის მიმართ. ODBC ტექნოლგია კარგად მუშაობს ტრადიციულ მონაცემთა ბაზებთან, მაგრამ ის არ იძლევა ისეთ მონაცემეთან მიმართვის შესაძლებლობას, რომლებიც არ არის მოცემული სტრიქონებისა და სვეტების სახით ან არ აქვს რეგულარული სტრუქტურა.

ამ პრობლემის გადასაჭრელად შეიმუშავეს OLE DB ტექნოლოგია, რომელიც ODBC-ის ანალოგიურად მუშაობს. ის შუალედური რგოლია კლიენტის პროგრამასა და მონაცემთა ბაზას შორის და კლიენტის პროგრამას მონაცემებს წარუდგენს ცხრილური სახით. ADO.NET უზრუნველყოფს როგორც OLE DB, ისე ODBC ტექნოლოგიას. შემდეგ, შემუშავდა ActiveX Data Objects (ADO) ტექნოლოგია. ADO პროგრამული სისტემაა, რომელიც მაღალი დონის ენებზე დაწერილ პროგრამებს საშუალებას აძლევს მიმართონ OLE DB მონაცემებს.

ADO.NET კლასები და ობიექტები

ADO.NET საბაზო კლასები შეგვიძლია ორ ნაწილად დავყოთ: კლასები, რომლებიც აღწერენ **ობიექტ-მომხმარებლებს** (ესაა ობიექტები, რომლებიც მონაცემებს იღებენ) და კლასები, რომლებიც აღწერენ **ობიექტ-მიმწოდებლებს** (ესაა ობიექტები, რომლებიც გასცემენ მონაცემებს) (ნახ. 16. 7). ობიექტი-მიმწოდებელი წაიკითხავს მონაცემებს მონაცემთა ბაზებიდან და მიაწოდებს მათ ობიექტ-მომხმარებელს. ამიტომ, ობიექტი-მიმწოდებელი ითხოვს სერვერთან აქტიური შეერთების არსებობას. ობიექტ-მომხმარებელს კი მონაცემების მიღების შემდეგ ამ მონაცემებთან მუშაობა შეუძლია ავტონომურ (ოფლაინ) რეჟიმში ანუ რეჟიმში, როცა მონაცემთა ბაზასთან კავშირი დახურულია.



ნახ. 16.7. ADO.NET საბაზო კლასები

ობიექტი-მომხმარებელი და შესაბამისი კლასები

მოკლედ აღვწეროთ ობიექტ-მომხმარებელთან სამუშაო კლასები.

DataSet კლასი. ამ კლასის ობიექტი გამოიყენება მონაცემთა ბაზაში მოთავსებული მონაცემების ლოკალური ასლების შესანახად კლიენტის კომპიუტერზე. ეს ობიექტებია: ცხრილები, ცხრილებს შორის კავშირები (ბმები), წარმოდგენები, შენახული პროცედურები და ფუნქციები. მათთან მუშაობა სრულდება ავტონომურ რეჟიმში. DataSet ობიექტები შეიძლება, აგრეთვე, გამოვიყენოთ მონაცემების წარმოსადგენად XML ფაილებიდან.

DataTable კლასი. ამ კლასის ობიექტი გამოიყენება ერთი ცხრილის, მაგალითად Personali

ცხრილის შესანახად. ერთ DataSet ობიექტში შეიძლება რამდენიმე DataTable ობიექტის ანუ რამდენიმე ცხრილის მოთავსება.

DataRow კლასი. ამ კლასის ობიექტი გამოიყენება ცხრილის ერთი სტრიქონის შესანახად. ერთ DataTable ობიექტში შეიძლება რამდენიმე DataRow ობიექტის ანუ რამდენიმე სტრიქონის მოთავსება.

DataColumn კლასი. ამ კლასის ობიექტი გამოიყენება ერთი სვეტის შესანახად. ერთ DataRow ობიექტში შეიძლება რამდენიმე DataColumn ობიექტის ანუ რამდენიმე სვეტის მოთავსება.

DataRelation კლასი. ამ კლასის ობიექტი გამოიყენება ორ DataTable ობიექტს ანუ ორ ცხრილს შორის კავშირის დასამყარებლად. DataRelation ობიექტები შეგვიძლია გამოვიყენოთ მონაცემთა ბაზაში ორ ცხრილს შორის "მთავარი-დამოკიდებული" კავშირის შესაქმნელად. ერთ DataSet ობიექტში შეიძლება რამდენიმე DataRelation ობიექტის მოთავსება.

Constraint კლასი. ამ კლასის ობიექტი გამოიყენება ცხრილის შეზღუდვების შესანახად, როგორიცაა სვეტის მნიშვნელობის შეზღუდვა უნიკალურობაზე ან გარე გასაღების შეზღუდვა, რომელიც სხვა ცხრილს მიმართავს. ერთ DataTable ობიექტში შეიძლება რამდენიმე Constraint ობიექტის მოთავსება.

DataView კლასი. ამ კლასის ობიექტი გამოიყენება DataTable ობიექტში მოთავსებული ცხრილის გაფილტვრისა და დახარისხებისათვის. ერთი DataSet ობიექტი შეიძლება რამდენიმე DataView ობიექტს შეიცავდეს.

DataSet, DataTable, DataRow, DataColumn, DataRelation, Constraint და **DataView** კლასები განსაზღვრულია **System::Data** სახელების სივრცეში.

ობიექტი-მიმწოდებელი და შესაბამისი კლასები

მოკლედ აღვწეროთ ობიექტ-მიმწოდებელთან სამუშაო კლასები.

SqlConnection, OleDbConnection, OdbcConnection კლასები. SqlConnection კლასის ობიექტი გამოიყენება SQL სერვერზე მონაცემთა ბაზასთან შეერთების დასამყარებლად. OleDbConnection კლასის ობიექტი გამოიყენება იმ მონაცემთა ბაზის მართვის სისტემასთან მისაერთებლად, რომელიც OLEDB ტექნოლოგიას (Object Linking and Embedding for Databases - ობიექტის დაკავშირება და ჩადგმა მონაცემთა ბაზებისათვის) უზრუნველყოფს. ასეთი სისტემებია Oracle და Access. OdbcConnection კლასის ობიექტი გამოიყენება იმ მონაცემთა ბაზის მართვის სისტემასთან მისაერთებლად, რომელიც ODBC (Open Database Connectivity - მონაცემთა ბაზასთან მიმართვის ღია ინტერფეისი) ტექნოლოგიას უზრუნველყოფს. ყველა ძირითადი მონაცემთა ბაზა უზრუნველყოფს ODBC ტექნოლოგიას, მაგრამ მისი საშუალებით მონაცემთა ბაზასთან მიმართვა .NET სისტემაში, ჩვეულებრივ, ნელა სრულდება, ვიდრე წინა ორი საშუალებით.

SqlCommand, OleDbCommand, OdbcCommand კლასები. ამ კლასების ობიექტები გამოიყენება SQL მოთხოვნის ფორმირებისათვის ან შენახული პროცედურის გამოძახებისათვის. ფორმირებული მოთხოვნა ან პროცედურის გამოძახება შეიძლება შესრულდეს წინა აბზაცში ჩამოთვლილი კლასებიდან ერთ-ერთის საშუალებით.

SqlDataReader, OleDbDataReader, OdbcDataReader კლასები. ამ კლასების ობიექტები გამოიყენება ინფორმაციის წასაკითხად მონაცემთა ბაზიდან, რომელიც მოთავსებულია SQL სერვერზე ან მუშაობს იმ მონაცემთა ბაზის მართვის სისტემის ქვეშ, რომელიც უზრუნველყოფს OLEDB ან ODBC ტექნოლოგიას. ეს ობიექტები გამოიყენება მონაცემების მხოლოდ წასაკითხად მონაცემების წყაროდან და წარმოადგენენ DataSet ობიექტის ალტერნატივას. მონაცემების წაკითხვა ამ ობიექტების საშუალებით, როგორც წესი, უფრო სწრაფად სრულდება, ვიდრე DataSet ობიექტის გამოყენებით.

SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter კლასები. ამ კლასების ობიექტები

გამოიყენება სტრიქონების გადასაადგილებლად DataSet ობიექტსა და მონაცემთა ბაზას შორის, რომელიც მოთავსებულია SQL სერვერზე, ან მუშაობს იმ მონაცემთა ბაზის მართვის სისტემის ქვეშ, რომელიც უზრუნველყოფს OLEDB ან ODBC ტექნოლოგიას.

მართვადი კლასები SQL სერვერისათვის გამოცხადებულია **System::Data::SqlClient** სახელების სივრცეში. კლასები მონაცემთა ბაზებისათვის, რომლებიც უზრუნველყოფენ ODBC ტექნოლოგიას, გამოცხადებულია **System::Data::Odbc** სახელების სივრცეში.

System.Data სახელების სივრცე

C++ პროგრამაში ADO.NET კლასების გამოყენებამდე using დირექტივების ბლოკში უნდა მოვათავსოთ დირექტივა:

```
using System::Data
```

სწორედ ამ სახელების სივრცეშია მოთავსებული ADO.NET კლასები. ამის შემდეგ, ჩვენ უნდა ავირჩიოთ .NET მონაცემების კონკრეტული მიმწოდებელი, ჩვენ მიერ გამოყენებული მონაცემთა კონკრეტული წყაროსთვის:

- თუ ვმუშაობთ SQL სერვერზე მოთავსებულ მონაცემთა ბაზასთან, მაშინ using დირექტივების ბლოკში უნდა მოვათავსოთ using System::Data::SqlClient დირექტივა.
- თუ ვიყენებთ SQL სერვერისა და Oracle-საგან განსხვავებული მონაცემთა ბაზას, მაგალითად Access, მაშინ using დირექტივების ბლოკში უნდა მოვათავსოთ using System::Data::OleDB დირექტივა.
- თუ ვიყენებთ მონაცემთა წყაროებს, რომლებისთვისაც არ არსებობს „მშობლიური“ OLE DB მომწოდებელი. using დირექტივების ბლოკში უნდა მოვათავსოთ using System::Data::Odbc დირექტივა.

მოთხოვნების შესრულება ADO.NET კლასების გამოყენებით

მონაცემების წაკითხვა DataReader ობიექტის საშუალებით

DataReader ობიექტი SELECT ბრძანებაში (მოთხოვნაში) მითითებული ცხრილიდან თითო სტრიქონს კითხულობს. წაკითხვის პროცესი შემდეგი მოქმედებებისაგან შედგება:

1. მონაცემების წყაროსთან მიერთება. ამისათვის, ვქმნით შეერთების ობიექტს:

```
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
```

myConnection ობიექტს ენიჭება შეერთების სტრიქონი, რომელიც ოთხ პარამეტრს შეიცავს: **კომპიუტერის სახელი**, რომელზეც მუშაობს SQL სერვერი. ამ სახელს მივუთითებთ შეერთების სტრიქონის server პარამეტრში. თუ SQL სერვერი ქსელურ კომპიუტერზეა მოთავსებული, მაშინ უნდა მივუთითოთ კომპიუტერის სახელი. თუ სერვერი ჩვენს (ლოკალურ) კომპიუტერზე მუშაობს, მაშინ შეგვიძლია მივუთითოთ როგორც კომპიუტერის სახელი, ისე localhost სიტყვა, server = localhost. **მონაცემთა ბაზის სახელი** database პარამეტრში ეთითება, database = Shekveta. **მომხმარებლის სახელი** შეერთების სტრიქონის uid პარამეტრში ეთითება, uid = sa. **პაროლი**, მონაცემთა ბაზის მოცემული მომხმარებლისათვის, pwd პარამეტრში ეთითება, pwd=paroli.

2. შეერთების გახსნა:

```
myConnection->Open();
```

3. SELECT ბრძანების ფორმირება:

```
myCommand->CommandText = "SELECT gvari, ganyofileba, asaki, tarigi_dabadebis FROM Personali";
```

ამ SELECT ბრძანების შესრულების შედეგად მიღებულ სტრიქონში მოთავსებული იქნება მხოლოდ მითითებული სვეტები. თუ გვინდა ყველა სვეტის მიღება, მაშინ SELECT სიტყვის შემდეგ სვეტების სახელების ნაცვლად უნდა მივუთითოთ '*'.

4. SqlDataReader ტიპის ობიექტის საშუალებით მონაცემების წაკითხვა და ეკრანზე მათი გამოტანა. myCommand ობიექტის ExecuteReader() მეთოდის საშუალებით იქმნება SqlDataReader ტიპის myReader ობიექტი. ExecuteReader() მეთოდი ასრულებს SELECT ბრძანებას და ქმნის ობიექტ-წამკითხველს გენერირებული შედეგების წაკითხვისათვის. ეს ობიექტი-წამკითხველი ენიჭება myReader ობიექტს.

ობიექტი-წამკითხველიდან შედეგების მიღების ერთ-ერთი გზაა myReader ობიექტის Read() მეთოდის გამოყენება. ეს მეთოდი კითხულობს ერთ სტრიქონს, რომელიც მიღებული იყო SELECT ბრძანების შესრულების შედეგად და გასცემს true-ს, თუ კიდევ არის წასაკითხი სტრიქონები, წინააღმდეგ შემთხვევაში - false-ს. კოდის ფრაგმენტს აქვს სახე:

```
SqlDataReader^ myReader = myCommand->ExecuteReader();
for ( ; myReader->Read(); )
        label1->Text += myReader["gvari"] + "   " + myReader["ganyofileba"] + "   " +
        myReader["asaki"]->ToString() + "   " + myReader["tarigi_dabadebis"] + "\n";
```

როგორც, კოდის ამ ფრაგმენტიდან ჩანს, ციკლი სრულდება მანამ, სანამ Read() მეთოდი გასცემს true მნიშვნელობას. ყურადღება მივაქციოთ იმას, რომ საჭირო სვეტთან მიმართვისთვის მისი სახელი მითითებულია myReader სახელის მარჯვნივ კვადრარულ ფრჩხილებში, მაგალითად, myReader["gvari"]. სვეტის სახელის ნაცვლად შეიძლება მისი ინდექსის მითითებაც - myReader[0]. "gvari" სვეტის ინდექსია 0, რადგან ეს პირველი სვეტია SELECT მოთხოვნაში, მეორეა "ganyofileba" სვეტი, ამიტომ მისის ინდექსია 1 და ა.შ.

5. DataReader-სა და შეერთების დახურვა:

```
myReader->Close();
myConnection->Close();
```

მოყვანილი პროგრამით ხდება განხილული ეტაპების რეალიზება, კერძოდ SQL სერვერის Shekveta ბაზის Personali ცხრილიდან სტრიქონების წაკითხვა:

```
//      პროგრამა 16.1
//      SQL სერვერის Shekveta ბაზის Personali ცხრილიდან სტრიქონების წაკითხვა
{
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ამ შეერთებისთვის ბრძანების შექმნა
SqlCommand^ myCommand = myConnection->CreateCommand();
//      SQL მოთხოვნის განსაზღვრა ამ ბრძანებისთვის
myCommand->CommandText = "SELECT gvari, ganyofileba, asaki, tarigi_dabadebis FROM Personali";
//      მოცემული ბრძანებისთვის DataReader-ის შესრულება
SqlDataReader^ myReader = myCommand->ExecuteReader();
//      კითხვა მანამ, სანამ არის წასაკითხი სტრიქონები
for ( ; myReader->Read(); )
label1->Text += myReader["gvari"] + "   " + myReader["ganyofileba"] + "   " +
myReader["asaki"]->ToString() + "   " + myReader["tarigi_dabadebis"] + "\n";
//      მკითხველის დახურვა
myReader->Close();
//      შეერთების დახურვა
myConnection->Close();
```

}

Access მონაცემთა ბაზიდან სტრიქონების წასაკითხად, ჩვენს პროგრამას დასაწყისში ჯერ დავუმატოთ

using System::Data::OleDb;

დირექტივა, შემდეგ კი 16.19 პროგრამაში SqlConnection, SqlCommand და SqlDataReader ობიექტები შევცვალოთ OleDbConnection, OleDbCommand და OleDbDataReader ობიექტებით. უნდა შევცვალოთ, აგრეთვე შეერთების სტრიქონი, რომელსაც ექნება შემდეგი სახე:

"Provider=Microsoft.Jet.OLEDB.4.0;
DataSource=C:\\Users\\Romani\\Documents\\ROMANI\\Access\\db5.mdb"

შეერთების სტრიქონში Provider კონსტრუქცია განსაზღვრავს OLE DB მიმწოდებელს Access მონაცემთა ბაზისთვის, DataSource კონსტრუქცია კი განსაზღვრავს მონაცემთა ბაზის კონკრეტულ ფაილს, რომელთანაც უნდა შესრულდეს მიმართვა.

მოცემული პროგრამით სრულდება Access მონაცემთა ბაზიდან, კერძოდ db5.mdb მონაცემთა ბაზის Personali ცხრილიდან სტრიქონების წაკითხვა:

```
{
//      პროგრამა 16.2
//      Access სისტემის Shekveta ბაზის Personali ცხრილიდან სტრიქონების წაკითხვა
label1->Text = "";
//      შეერთების ობიექტის შექმნა
OleDbConnection^ myConnection = gcnew OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\\Users\\Romani\\Documents\\ROMANI\\Access\\db5.mdb");
//      შეერთების გახსნა
myConnection->Open();
//      ბრძანების შექმნა ამ შეერთებისთვის
OleDbCommand^ myCommand = myConnection->CreateCommand();
//      SQL მოთხოვნის განსაზღვრა ამ ბრძანებისთვის
myCommand->CommandText = "SELECT gvari, ganyofileba, asaki, tarigi_dabadebis FROM Personali";
//      მოცემული ბრძანებისთვის DataReader-ის შესრულება
OleDbDataReader^ myReader = myCommand->ExecuteReader();
//      სანამ არის წასაკითხი სტრიქონები
for ( ; myReader->Read(); )
label1->Text += myReader["gvari"] + "   " + myReader["ganyofileba"] + "   " +
            myReader["asaki"]->ToString() + "   " + myReader["tarigi_dabadebis"] + "\n";
//      მკითხველის დახურვა
myReader->Close();
//      შეერთების დახურვა
myConnection->Close();
}
```

მონაცემთა ბაზასთან მუშაობა DataSet ობიექტის გამოყენებით

მონაცემების კითხვა

ახლა ვნახოთ თუ როგორ შეიძლება მონაცემების წაკითხვა მონაცემთა ბაზიდან DataSet ობიექტის გამოყენებით. როგორც 16.8 ნახაზიდან ჩანს, DataSet ობიექტი მოიცავს DataRelation და DataTable ობიექტებს. თავის მხრივ, DataTable ობიექტი მოიცავს DataRow ობიექტებს, DataRow ობიექტი კი – DataColumn ობიექტებს. აქედან, გამომდინარე, DataSet ობიექტში შეგვიძლია მოვათავსოთ ერთი ან მეტი ცხრილი, მაგალითად Personali, Shemkveti და Xelshekruleba, და მათ

შორის ბმები (რელაციები, კავშირები). მხოლოდ ამის შემდეგ შეგვეძლება ამ ცხრილებთან მუშაობა. ცხრილებით DataSet ობიექტის შესავსებად გამოიყენება ადაპტერის Fill() მეთოდი. ადაპტერი არის ობიექტი, რომელიც გამოიყენება მონაცემების გადასაგზავნად კლიენტიდან (ჩვენი კომპიუტერიდან) სერვერზე და პირიქით.

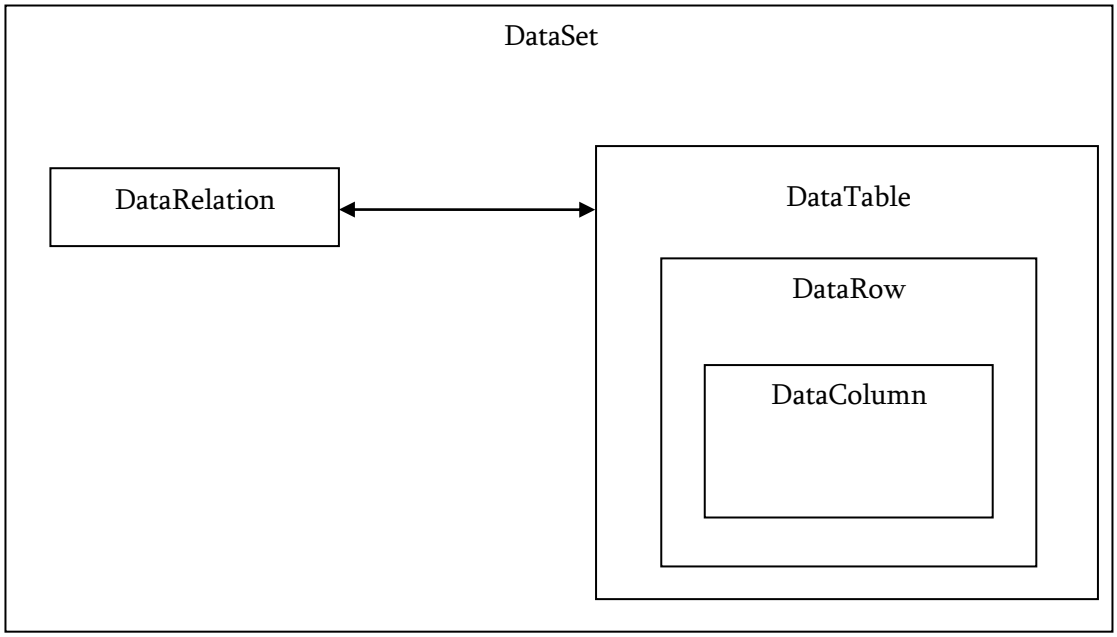
DataSet ობიექტს აქვს Tables თვისება, რომელიც წარმოადგენს DataTable ტიპის ობიექტების კოლექციას. ეს თვისება საშუალებას გვაძლევს ცხრილებს მივმართოთ როგორც სახელის, ისე ინდექსის მიხედვით. მაგალითად,

```
myDataSet.Tables["Personali"];  ან
myDataSet.Tables[0];
```

შედეგად, DataTable ობიექტში შეგვიძლია მოვათავსოთ არჩეული ცხრილი:

```
DataTable personaliTable = new DataTable();
personaliTable = myDataSet.Tables["Personali"];
```

შევნიშნოთ, რომ ცხრილების ნუმერაცია (ინდექსაცია) ნულიდან იწყება.



ნახ. 16.8. DataSet ობიექტის სტრუქტურა

Tables თვისებას, თავის მხრივ აქვს Rows თვისება, რომელიც არის DataRow ობიექტების კოლექცია. ეს თვისება საშუალებას გვაძლევს ცხრილის სტრიქონს მივმართოთ ინდექსის მიხედვით. მაგალითად,

```
myDataSet->Tables["Personali"]->Rows[2];
```

შედეგად, DataRow ობიექტში შეგვიძლია მოვათავსოთ არჩეული სტრიქონი:

```
DataRow^ myRow = myDataSet->Tables["Personali"]->Rows[3];
```

შევნიშნოთ, რომ სტრიქონების ინდექსაცია 0-დან იწყება.

Rows თვისება საშუალებას გვაძლევს, აგრეთვე მივმართოთ სტრიქონის სვეტებს სახელის ან ინდექსის მიხედვით. მაგალითად,

```
Object^ gvari_1 = myDataSet->Tables["Personali"]->Rows[2]["gvari"];
Object^ gvari_2 = myDataSet->Tables["Personali"]->Rows[2][1];
```

შეგვიძლია, აგრეთვე კონკრეტულ ცვლადში ჩავწეროთ არჩეული სვეტის მნიშვნელობა

```
String^ gvari_3 = myDataset->Tables["Personali"]->Rows[2]["gvari"]->ToString();
```

უნდა გვახსოვდეს, რომ სვეტების ნუმერაცია 0-დან იწყება. რაც შეეხება იმ საკითხს,

სახელის გამოყენება ჯობია თუ ინდექსის, უმჯობესია სახელის გამოყენება, რადგან მომავალში ცხრილში შეიძლება სვეტების მიმდევრობა შეიცვალოს.

მოცემული პროგრამით ხდება სერვერზე მოთავსებული Shekveta მონაცემთა ბაზის Personali ცხრილის სტრიქონების წაკითხვა და მათი მოთავსება myDataSet ობიექტში:

```
{
//      პროგრამა 16.3
//
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew SqlConnection("server=ROMANI-PC; database=Shekveta;
uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      DataAdapter ობიექტის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT  gvari, ganyofileba, asaki,
tarigi_dabadebis FROM Personali", myConnection);
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset,"Personali");
//      Personali ცხრილის სტრიქონების გამოტანა ეკრანზე
dataGridView1->DataSource = myDataset;
dataGridView1->DataMember = "Personali";
//      შეერთების დახურვა
myConnection->Close();
}
```

სტრიქონში:

```
myAdapter->Fill(myDataset,"Personali");
```

myDataset ობიექტში იქმნება DataTable ტიპის Personali ობიექტი, რომელიც ივსება სტრიქონებით Shekveta მონაცემთა ბაზის Personali ცხრილიდან. Fill() მეთოდი ასრულებს myAdapter ობიექტთან დაკავშირებულ SELECT მოთხოვნას. ამრიგად, Personali ობიექტი მოთავსებულია არა სერვერზე, არამედ კლიენტის კომპიუტერზე (ჩვენს კომპიუტერზე) myDataset ობიექტში. კლიენტისა და სერვერის სტრუქტურა ნაჩვენებია 16. 9 ნახაზზე.

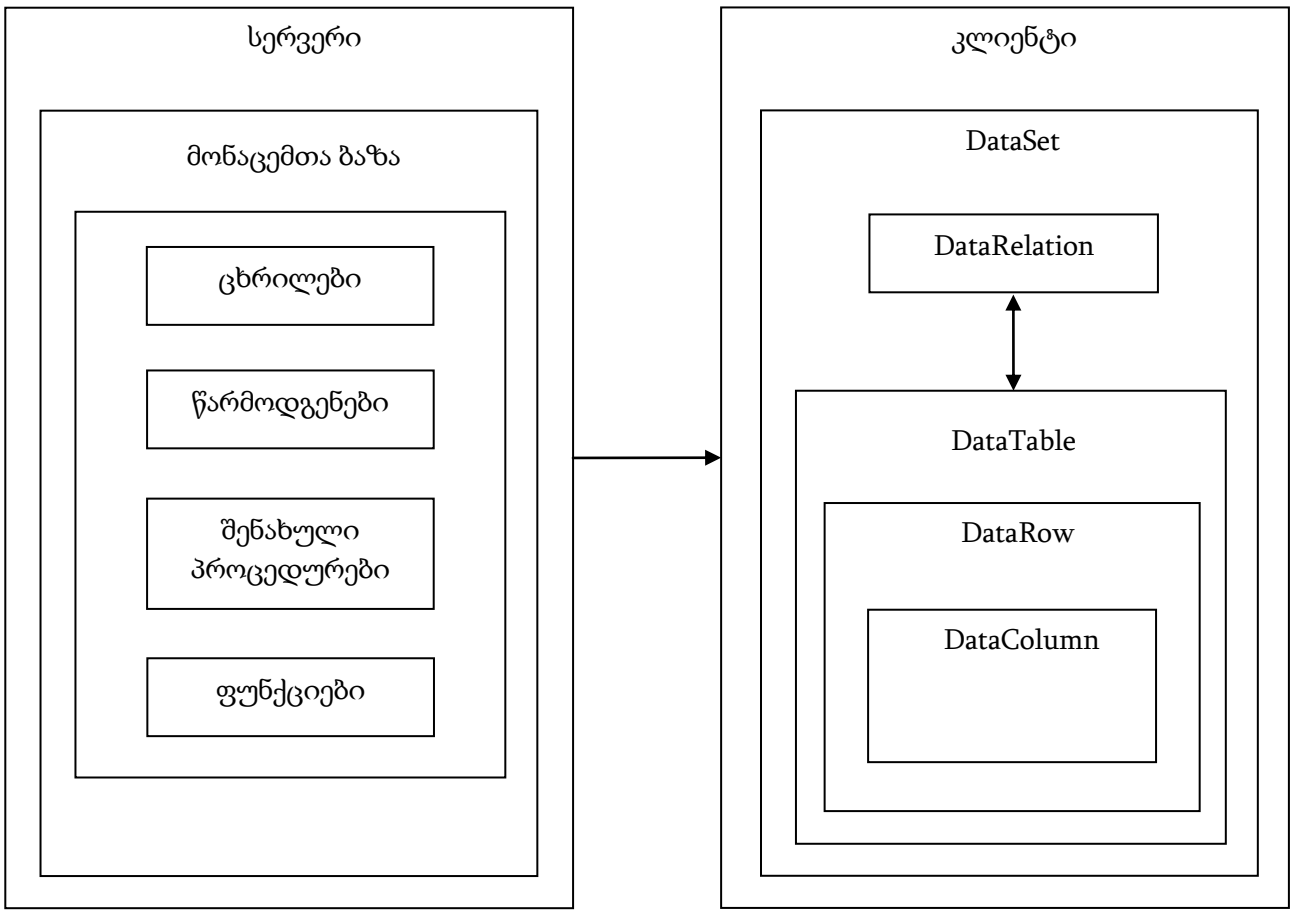
ცხრილებით DataSet ობიექტის შევსების შემდეგ, ამ ცხრილებთან მუშაობა შეგვეძლება ავტონომიურ (ოფლაინ) რეჟიმში ანუ სერვერთან კავშირის გარეშე.

მონაცემთა ბაზასთან შეერთების აშკარად გახსნა ან დახურვა აუცილებელი არ არის, რადგან ამას ადაპტერი ავტომატურად აკეთებს. myAdapter ობიექტი-ადაპტერი, საჭიროების მიხედვით, ხსნის და ხურავს შეეთებას მონაცემთა ბაზასთან. ადაპტერი შეერთების მდგომარეობას უცვლელად ინარჩუნებს. ამიტომ, თუ შეერთება გახსნილი იყო ადაპტერის მუშაობის დაწყებამდე, მაშინ ადაპტერის მუშაობის დამთავრების შემდეგ შეერთება ისევ გახსნილი დარჩება. ამისგან, განსხვავებით, DataReader ითხოვს მონაცემთა ბაზასთან შეერთების არსებობას მისი მუშაობის მთელი პერიოდის განმავლობაში.

Personali ცხრილის სტრიქონები შეგვიძლია გამოვიტანოთ label კომპონენტშიც:

```
for each ( DataRow^ theRow in myDataset->Tables["Personali"]->Rows )
label1->Text += theRow["gvari"] + "  " + theRow["ganyofileba"] + "   " +
                        theRow["asaki"]->ToString() + "  " + theRow["tarigi_dabadebis"] + "\n";
```

თუმცა dataGridView კომპონენტში ცხრილის სტრიქონების გამოტანა გაცილებით სწრაფად სრულდება.



ნახ. 16. 9. კლიენტისა და სერვერის სტრუქტურა

თუ ადაპტერის შექმნისას SELECT ბრძანებას აქვს შემდეგი სახე:

```
SqlDataAdapter^ personaliAdapter =
                                    gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
```

მაშინ Fill() მეთოდის შესრულებისას გაიცემა Personali ცხრილის ყველა სვეტი და ყველა სტრიქონი. ასეთი მიდგომა შეგვიძლია გამოვიყენოთ მცირე ზომის ცხრილებისათვის, რომლებშიც სვეტებისა და სტრიქონების რაოდენობა მცირეა. როცა ცხრილში სტრიქონების რაოდენობა აღემატება 100 000-ს, მაშინ ასეთი მიდგომა კარგად ვერ იმუშავებს. ასეთ შემთხვევებში, საჭიროა გამოვიყენოთ როგორც ვერტიკალური, ისე ჰორიზონტალური ფილტრი. ვერტიკალური ფილტრი გვექნება მაშინ, როცა SELECT ბრძანებაში მივუთითებთ ცხრილის მხოლოდ იმ სვეტებს, რომლებთანაც რეალურად ვაპირებთ მუშაობას, მაგალითად:

```
SqlDataAdapter^ personaliAdapter = gcnew SqlDataAdapter(
                                    "SELECT gvari, staji, xelfasi FROM Personali", myConnection);
```

მაგრამ, ასეთი მიდგომა არ გამოდგება იმ შემთხვევაში, როცა ცხრილს სტრიქონებს ვუმატებთ, რადგან ამ დროს სტრიქონის თითოეულ სვეტს უნდა მივანიჭოთ შესაბამისი მნიშვნელობა.

ჰორიზონტალური ფილტრის შემთხვევაში, SELECT ბრძანებაში უნდა მივუთითოთ WHERE განყოფილება. შედეგად, ამოირჩევა ცხრილის ერთი ან მეტი სტრიქონი, რომლებთანაც რეალურად ვაპირებთ მუშაობას. დავუშვათ, გვინდა იმ თანამშრომლებზე ინფორმაცია, რომელთა სტაჟი აღემატება 20 წელს. მაშინ SELCET მოთხოვნას ენება სახე:

```
SELECT * FROM Personali WHERE staji >= 20
```

შეგვიძლია მივუთითოთ სვეტებიც:

```
SELECT gvari, asaki, staji, xelfasi FROM Personali WHERE staji >= 20
```

ცუდ მიდგომად ითვლება მთელი ცხრილის გადმოწერა სერვერიდან DataSet ობიექტში და შემდეგ ამ ობიექტში ძებნის შესრულება. ასეთ დროს უმჯობესია SELECT ბრძანების გამოყენება WHERE კონსტრუქციასთან ერთად.

როგორც ვიცით, DataSet ობიექტში შეგვიძლია რამდენიმე ცხრილის მოთავსება. მოყვანილი პროგრამით ხდება myDataSet ობიექტში Personali, Shemkveti და Xelshekruleba ცხრილების მოთავსება:

```
{
//      პროგრამა 16.4
//      პროგრამით ხდება Personali, Shemkveti და Xelshekruleba ცხრილების
//      მოთავსება myDataset მონაცემთა ნაკრებში
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      DataAdapter ობიექტების შექმნა თითოეული ცხრილისთვის
SqlDataAdapter^ personaliAdapter =
                        gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
SqlDataAdapter^ shemkvetiAdapter =
                        gcnew SqlDataAdapter("SELECT * FROM Shemkveti", myConnection);
SqlDataAdapter^ xelshekrulebaAdapter =
                        gcnew SqlDataAdapter("SELECT * FROM Xelshekruleba", myConnection);
//      ცხრილების შევსება
personaliAdapter->Fill(myDataset, "Personali");
shemkvetiAdapter->Fill(myDataset, "Shemkveti");
xelshekrulebaAdapter->Fill(myDataset, "Xelshekruleba");
//      ეკრანზე ცხრილების გამოტანა
dataGridView1->DataSource = myDataset;
dataGridView1->DataMember = "Personali";
dataGridView2->DataSource = myDataset;
dataGridView2->DataMember = "Shemkveti";
dataGridView3->DataSource = myDataset;
dataGridView3->DataMember = "Xelshekruleba";
//      შეერთების დახურვა
myConnection->Close();
}
```

მონაცემების გაფილტვრა

მონაცემების გასაფილტრად შეგვიძლია DataView ობიექტის გამოყენება. დავუშვათ, გვაინტერესებს ინფორმაცია სამედიცინო განყოფილებაში მომუშავე იმ თანამშრომლების შესახებ, რომელთა ასაკი აღემატება 30 წელს. მოცემული კოდით ხდება ამის დემონსტრირება:

```
{
//      პროგრამა 16.5
//      პროგრამით ხდება Personali ცხრილის სტრიქონების გაფილტვრა
//      dataView ობიექტის გამოყენებით
```

```
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew SqlConnection("server=ROMANI-PC; database=Shekveta;
uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        DataAdapter ობიექტის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//        DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//        myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//        შეერთების დახურვა
myConnection->Close();
//        dataView1 ობიექტის დაკავშირება Personali ცხრილთან
DataView^ dataView1 = gcnew DataView(myDataset->Tables["Personali"]);
//        სტრიქონების გაფილტვრა
dataView1->RowFilter = L"ganyofileba = 'სამედიცინო' AND asaki > 50";
//        გაფილტრული სტრიქონების ასახვა ეკრანზე
dataGridView1->DataSource = dataView1;
}
```

როგორც პროგრამიდან ჩანს dataView1 ობიექტის RowFilter თვისებას უნდა მივანიჭოთ ფილტრი, რომელიც სტრიქონს წარმოადგენს. ფილტრის გასაუქმებლად უნდა შევასრულოთ კოდი:

```
dataView1->RowFilter = "";
```

ფილტრის შესატანად შეგვიძლია textBox კომპონენტის გამოყენებაც:

```
dataView1->RowFilter = textBox1->Text;
```

ამ შემთხვევაში, textBox კომპონენტში ფილტრი შეგვაქვს ბრჭყალების გარეშე.

თუ გვაინტერესებს ის თანამშრომლები, რომლებიც დაიბადნენ 1990 წლის 1 იანვრის შემდეგ, მაშინ dataView1 ობიექტის RowFilter თვისებას უნდა მივანიჭოთ შემდეგი ფილტრი:

```
dataView1->RowFilter = L"tarigi_dabadebis > '01.01.1990'"
```

თარიღის შეტანა შეგვიძლია dateTimePicker1 კომპონენტიდანაც:

```
dataView1->RowFilter = L"tarigi_dabadebis > '" + dateTimePicker1->Value.ToString() + "'";
```

მონაცემების დახარისხება

მონაცემების დასახარისხებლად შეგვიძლია DataView ობიექტის გამოყენება. მოყვანილი კოდით ხდება Personali ცხრილის სტრიქონების დახარისხება ganyofileba სვეტის მნიშვნელობების ზრდის მიხედვით და asaki სვეტის მნიშვნელობების კლების მიხედვით:

```
{
//        პროგრამა 16.6
//        პროგრამით ხდება Personali ცხრილის სტრიქონების დახარისხება
//        dataView ობიექტის გამოყენებით
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        DataAdapter ობიექტის შექმნა
```

```
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//        DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//        myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//        შეერთების დახურვა
myConnection->Close();
//        dataView1 ობიექტის დაკავშირება Personali ცხრილთან
DataView^ dataView1 = gcnew DataView(myDataset->Tables["Personali"]);
//        სტრიქონების დახარისხება
dataView1->Sort = "ganyofileba, asaki DESC";
//        დახარისხებული სტრიქონების ასახვა ეკრანზე
dataGridView1->DataSource = dataView1;
}
```

დახარისხების გაუქმება შემდეგნაირად შეგვიძლია:

```
dataView1->Sort = "";
```

სტრიქონების ძებნა

ცხრილში სტრიქონის მისაძებნად Find() მეთოდი გამოიყენება. ეს მეთოდი ითხოვს პირველადი გასაღების დაყენებას. საჭირო მნიშვნელობა იძებნება პირველად გასაღებში. პირველადი გასაღები შეიძლება შედგებოდეს ერთი ან მეტი სვეტისგან, რომელთა მნიშვნელობები ცალსახად განსაზღვრავენ ერთ სტრიქონს. ამიტომ, ძებნის შედეგი ყოველთვის იქნება ცხრილის ერთი სტრიქონი.

მოყვანილი პროგრამით ხდება Personali ცხრილში სტრიქონის ძებნის დემონსტრირება:

```
{
//        პროგრამა 16.7
//        პროგრამით ხდება Personali ცხრილში სტრიქონის ძებნა
label1->Text = "";
//        შეგვაქვს პირველადი გასაღების მნიშვნელობა
int find_key = Convert::ToInt32(textBox1->Text);
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew SqlConnection("server=ROMANI-PC; database=Shekveta;
uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        ადაპტერის მომზადება სამუშაოდ
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//        SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//        DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
 //        myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//        გასაღებების მასივის გამოცხადება პირველადი გასაღების განსაზღვრისათვის
array<DataColumn^>^ keys = gcnew array<DataColumn^>(1);
keys[0] = myDataset->Tables["Personali"]->Columns["PersonaliID"];
myDataset->Tables["Personali"]->PrimaryKey = keys;
```

```
//        findRow ობიექტს ენიჭება ნაპოვნი სტრიქონი
DataRow^ findRow = myDataset->Tables["Personali"]->Rows->Find(find_key);
//        თუ სტრიქონი მოიძებნა, მისი სვეტები გამოგვაქვს ეკრანზე
if ( findRow->IsNull("PersonaliID") == false )
   label1->Text = L"სტრიქონი მოიძებნა:   " + findRow["gvari"] + "   " + findRow["xelfasi"]->ToString();
else label1->Text = L"სტრიქონი ვერ მოიძებნა";
//        შეერთების დახურვა
myConnection->Close();
}
```

მოყვანილ პროგრამაში საძებნი მნიშვნელობა textBox1.Text კომპონენტიდან ენიჭება find_key ცვლადს. პირველადი გასაღებია Personali ცხრილის PersonaliID სვეტი. მისი განსაზღვრა შემდეგნაირად ხდება:

```
array<DataColumn^>^ keys = gcnew array<DataColumn^>(1);
keys[0] = myDataset->Tables["Personali"]->Columns["PersonaliID"];
myDataset->Tables["Personali"]->PrimaryKey = keys;
```

როგორც ვხედავთ, ჯერ იქმნება DataColumn ტიპის მქონე keys ობიექტი, რომელშიც უნდა მოვათავსოთ პირველადი გასაღები. რადგან პირველადი გასაღები შეიძლება იყოს ერთი ან მეტი სვეტი, ამიტომ keys არის DataColumn ტიპის ობიექტების მასივი. შემდეგ, ამ მასივის პირველ ელემენტს ენიჭება Personali ცხრილის PersonaliID სვეტი. ბოლოს კი - DataTable ტიპის Personali ობიექტის PrimaryKey თვისებას ენიჭება keys ობიექტი.

პირველადი გასაღების განსაზღვრის შემდეგ შეგვიძლია შევასრულოთ ძებნა Rows თვისების Find() მეთოდის გამოყენებით. ეს მეთოდი გასცემს DataRow ობიექტს (სტრიქონს), რომელიც ენიჭება ამავე ტიპის findRow ობიექტს. მეთოდს არგუმენტად გადაეცემა საძებნი სიდიდე. თუ პირველადი გასაღები ორი და მეტი სვეტისგან შედგება, მაშინ Find() მეთოდს არგუმენტად უნდა გადავცეთ ობიექტების მასივი. ჩვენს შემთხვევაში, პირველადი გასაღები ერთი სვეტისგან შედგება, ამიტომ მეთოდს ერთ მნიშვნელობას გადავცემთ. საძებნი მნიშვნელობა find_key ცვლადს ენიჭება textBox1 კომპონენტიდან პროგრამის დასაწყისში.

Find() მეთოდი სტრიქონის პოვნის შემთხვევაში ამ სტრიქონს გასცემს DataRow ობიექტის სახით, წინააღმდეგ შემთხვევაში, გასცემს null მიმართვას. ამ შემოწმებას ასრულებს if ოპერატორი:

```
if ( findRow->IsNull("PersonaliID") == false )
```

ასეთივე გზით შეგვიძლია ჯერ მოვძებნოთ საჭირო სტრიქონი და შემდეგ შევცვალოთ მისი სვეტები ან წავშალოთ ნაპოვნი სტრიქონი.

ცხრილში მონაცემების შეცვლა

ამ განყოფილებაში ვნახავთ თუ როგორ შეიძლება შევცვალოთ ცხრილის მონაცემები. ამისათვის უნდა შევასრულოთ შემდეგი მოქმედებები:

- DataSet ობიექტის შევსება იმ ცხრილებით, რომლებშიც გვინდა მონაცემების შეცვლა.
- DataSet ობიექტში ცხრილების შეცვლა.
- DataSet ობიექტში მოთავსებული ცხრილების გადაგზავნა სერვერზე მონაცემთა ბაზაში.

მოცემული პროგრამით ხდება Personali ცხრილის მე-4 სტრიქონში (მისი უნდექსია 3) "asaki" სვეტის მნიშვნელობის შეცვლა. SELECT ბრძანებაში პირველადი გასაღების (personaliID) მითითება აუცილებელია.

```
{
//        პროგრამა 16.8
//        პროგრამით ხდება Personali ცხრილის სტრიქონში სვეტის მნიშვნელობის შეცვლა
label1->Text = ""; label2->Text = "";
```

```
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew
                SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection.Open();
//        ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter(
    "SELECT  personaliID, gvari, ganyofileba, asaki, tarigi_dabadebis FROM Personali", myConnection);
//        SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//        DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//        myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//        მონაცემების გამოტანა ცვლილებამდე
label1->Text = myDataset->Tables["Personali"]->Rows[3]["asaki"]->ToString() + "\n";
//        Personalis ცხრილის მე-3 სტრიქონში ასაკის შეცვლა
myDataset->Tables["Personali"]->Rows[3]["asaki"] = Convert::ToInt32(textBox1->Text);
//        Update ბრძანების გამოძახება ცვლილებების დაფიქსირებისათვის ცხრილში
myAdapter->Update(myDataset, "Personali");
//        ეკრანზე ცვლილებების გამოტანა
label2->Text = myDataset->Tables["Personali"]->Rows[3]["asaki"]->ToString();
//        შეერთების დახურვა
myConnection->Close();
}
```

პროგრამაში ადაპტერის შექმნის შემდეგ უნდა შევქმნათ კორექტული SQL ბრძანებები: INSERT, UPDATE და DELETE, ცხრილებში მონაცემების შეცვლის მიზნით. ამას, ავტომატურად აკეთებს SqlCommandBuilder კლასი. მის კონსტრუქტორს არგუმენტად გადაეცემა myAdapter ობიექტი. myBuilder ობიექტის შექმნისას ხდება SQL ბრძანებების გენერირება და ასოცირება (დაკავშირება) myAdapter ობიექტთან კონსტრუქტორის მიერ.

myAdapter ობიექტის Update() მეთოდი ასრულებს myDataset ობიექტიდან Personali ცხრილის გადაგზავნას სერვერზე Shekveta მონაცემთა ბაზაში. ეს მეთოდი ამოწმებს Personali ცხრილის თითოეულ სტრიქონს, შეიცვალა თუ არა ის. Rows კოლექციის თითოეულ სტრიქონს აქვს RowState თვისება, რომელიც განსაზღვრავს სტრიქონი იყო თუ არა წაშლილი, დამატებული, შეცვლილი თუ დარჩა უცვლელი. თითოეული ცვლილება აისახება Update() მეთოდის მიერ მონაცემთა ბაზაში. უნდა გვახსოვდეს, რომ Update() მეთოდი ითხოვს ცხრილში პირველადი გასაღების არსებობას.

როგორც პროგრამიდან ჩანს, იმისათვის, რომ ცხრილის რომელიმე სტრიქონის ველს მივანიჭოთ საჭირო მნიშვნელობა, უნდა მივუთითოთ სტრიქონის ინდექსი (ნომერი) და სვეტის სახელი ან ინდექსი:

```
myDataset->Tables["Personali"]->Rows[3]["asaki"] = Convert::ToInt32(textBox1->Text);
```

ეს ცვლილება ეხება ჩვენს ლოკალურ კომპიუტერში მოთავსებულ myDataset ობიექტის Personali ცხრილს და არა სერვერზე მოთავსებული Shekveta მონაცემთა ბაზის Personali ცხრილს.

დავუბრუნდეთ სტრიქონს:

```
label2.Text = myDataset->Tables["Personali"]->Rows[3]["asaki"]->ToString();
```

ეს სტრიქონი ეკვივალენტურია შემდეგი სტრიქონების:

```
DataTable^ personaliTable = myDataset->Tables["Personali"];
```

```
DataRow^ myRow = personaliTable->Rows[3];
Object^ asaki = myRow["asaki"];
label2->Text = asaki->ToString();
```

რომელ კოდს გამოვიყენებთ, ჩვენი გადასაწყვეტია. ზოგადად, უმჯობესია იმ კოდის გამოყენება, რომელიც ჩვენთვის უფრო გასაგებია.

სტრიქონის სვეტებისთვის მნიშვნელობების მისანიჭებლად, მოცემული სტრიქონების ნაცვლად:

```
myDataset->Tables["Personali"]->Rows[3]["gvari"] = textBox1->Text;
myDataset->Tables["Personali"]->Rows[3]["asaki"] = Convert::ToInt32(textBox2->Text);
myDataset->Tables["Personali"]->Rows[3]["ganyofileba"] = textBox3->Text;
myDataset->Tables["Personali"]->Rows[3]["tarigi_dabadebis"] = dateTimePicker1->Value;
```

შეგვიძლია, შემდეგი კოდის გამოყენებაც:

```
DataTable^ personaliTable = myDataset->Tables["Personali"];
DataRow^ myRow = personaliTable->Rows[3];
myRow["gvari"] = textBox1->Text;
myRow["asaki"] = Convert::ToInt32(textBox2->Text);
myRow["ganyofileba"] = textBox3->Text;
myRow["tarigi_dabadebis"] = dateTimePicker1->Value;
```

SELECT, UPDATE, INSERT და DELETE ბრძანებების ნახვა

CommandBuilder ობიექტი, ახდენს SQL ბრძანებების გენერირებას, რომლებიც გამოიყენება მონაცემების მოდიფიცირებისთვის. ეს ბრძანებებია: INSERT, UPDATE და DELETE. მათი გენერირება ხდება ადაპტერში მითითებული SELECT ბრძანების საფუძველზე. გენერირებული ბრძანებების მისაღებად და სანახავად შეგვიძლია შესაბამისად გამოვიყენოთ CommandBuilder ობიექტის GetInsertCommand(), GetUpdateCommand() და GetDeleteCommand() მეთოდები. მოყვანილი პროგრამით ხდება ამის დემონსტრირება:

```
{
//      პროგრამა 16.9
//      პროგრამით ხდება INSERT, UPDATE და DELETE ბრძანებების კოდის გამოტანა ეკრანზე
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew
            SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT  personaliID, gvari FROM Personali",
myConnection);
//      SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
label1->Text += myAdapter->SelectCommand->CommandText + "\n";
//      UPDATE ბრძანების ეკრანზე გამოტანა
SqlCommand^ updateCommand = myBuilder->GetUpdateCommand();
label1->Text += updateCommand->CommandText + "\n";
//      INSERT  ბრძანების ეკრანზე გამოტანა
SqlCommand^ insertCommand = myBuilder->GetInsertCommand();
label1->Text += insertCommand->CommandText + "\n";
```

```
//        DELETE  ბრძანების ეკრანზე გამოტანა
SqlCommand^ deleteCommand = myBuilder->GetDeleteCommand();
label1->Text += deleteCommand->CommandText + "\n";
//        შეერთების დახურვა
myConnection->Close();
}
```

შედეგს ექნება სახე:

```
SELECT  personaliID, gvari FROM Personali
UPDATE [Personali] SET [gvari] = @p1 WHERE (([personaliID] = @p2) AND ((@p3 = 1 AND [gvari] IS NULL) OR ([gvari] = @p4)))
INSERT INTO [Personali] ([gvari]) VALUES (@p1)
DELETE FROM [Personali] WHERE (([personaliID] = @p1) AND ((@p2 = 1 AND [gvari] IS NULL) OR ([gvari] = @p3)))
```

ცხრილში სტრიქონების დამატება

ცხრილში ახალი სტრიქონების დასამატებლად შემდეგი მოქმედებები უნდა შევასრულოთ:

- ახალი DataRow ობიექტის შექმნა.
- მონაცემებით მისი შევსება.
- შევსებული სტრიქონის დამატება DataSet ობიექტის Rows კოლექციისთვის.
- ადაპტერის Update() მეთოდის შესრულება.

მოცემული პროგრამით ხდება Personali ცხრილში ახალი სტრიქონის დამატება:

```
{
//        პროგრამა 16.10
//        პროგრამით ხდება Personali ცხრილში ახალი სტრიქონის დამატება
label1->Text = "";
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew
                        SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter(
        "SELECT  personaliID, gvari, xelfasi, staji, tarigi_dabadebis FROM Personali", myConnection);
//        SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//        DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//        myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//        სტრიქონების რაოდენობა ცვლილებამდე
label1->Text = myDataset->Tables["Personali"]->Rows->Count.ToString() + "\n";
//        ახალი სტრიქონის შექმნა
DataRow^ myRow = myDataset->Tables["Personali"]->NewRow();
//        ახალი სტრიქონის სვეტებისთვის მნიშვნელობების მინიჭება
myRow["gvari"] = textBox1->Text;
myRow["staji"] = textBox2->Text;
myRow["xelfasi"] = textBox3->Text;
myRow["tarigi_dabadebis"] = dateTimePicker1->Value.ToString();
myDataset->Tables["Personali"]->Rows->Add(myRow);
```

```
//        სტრიქონების რაოდენობა ცვლილების შემდეგ
label1->Text += myDataset->Tables["Personali"]->Rows->Count.ToString() + "\n";
//        Update ბრძანების გამოძახება ცვლილებების დაფიქსირებისათვის ცხრილში
myAdapter->Update(myDataset, "Personali");
//        შეერთების დახურვა
myConnection->Close();
}
```

როგორც პროგრამის მოცემული სტრიქონიდან ჩანს:

```
label1->Text = myDataset->Tables["Personali"]->Rows->Count.ToString() + "\n";
```

Rows კოლექციას აქვს Count თვისება, რომელშიც ავტომატურად იწერება Personali ცხრილში არსებული სტრიქონების რაოდენობა. ამ ცხრილში ახალი სტრიქონის დამატების შემდეგ, ეს მნიშვნელობა ერთით გაიზრდება.

შემდეგ, იქმნება DataRow ტიპის myRow ობიექტი myDataset ობიექტის NewRow() მეთოდის გამოყენებით. myRow ობიექტის სტრუქტურა ემთხვევა Personali ცხრილის სტრუქტურას ანუ შეიცავს ამ ცხრილის მხოლოდ იმ სვეტებს, რომლებიც ადაპტერში განსაზღვრულ SELECT ბრძანებაშია მოთავსებული. ეს სვეტები ცარიელია და მათ უნდა მივანიჭოთ კონკრეტული მნიშვნელობები. რადგან, ცხრილის თითოეულ სვეტს აქვს object ტიპი, ამიტომ, მაგალითად "staji" სვეტს მნიშვნელობა შეგვიძლია მივანიჭოთ ან ასე:

```
myRow["staji"] = textBox2->Text;
```

ან ასე:

```
myRow["staji"] = Convert::ToDouble(textBox2->Text);
```

შედეგი ერთი და იგივე იქნება.

როცა ყველა სვეტს მივანიჭებთ საჭირო მნიშვნელობებს, მხოლოდ ამის შემდეგ უნდა დავუმატოთ myRow ობიექტი Personali ცხრილს Rows კოლექციის Add() მეთოდის გამოყენებით:

```
myDataset->Tables["Personali"]->Rows->Add(myRow);
```

ამის შემდეგ, ვასრულებთ ადაპტერის Update() მეთოდს ცვლილებების შესანახად სერვერზე მოთავსებულ მონაცემთა ბაზაში.

პროგრამაში მნიშვნელობები მიენიჭა Personali ცხრილის personaliID, gvari, xelfasi, staji, tarigi_dabadebis სვეტებს. ამ ცხრილის დანარჩენ სვეტებში ჩაიწერება null მნიშვნელობა. იმისათვის, რომ სხვა სვეტებსაც მივანიჭოთ მნიშვნელობები საჭიროა, რომ ამ სვეტების სახელები მოვათავსოთ SELECT მოთხოვნაში, რომელსაც მივუთითებთ ადაპტერის შექმნისას. მაგალითად, SELECT მოთხოვნას დავუმატოთ ganyofileba სვეტი:

```
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter(
"SELECT  personaliID, gvari, xelfasi, staji, tarigi_dabadebis, ganyofileba FROM Personali",
myConnection);
```

ამის შემდეგ, შეგვიძლია აღნიშნულ სვეტს მივანიჭოთ მნიშვნელობა:

```
myRow["ganyofileba"] = Convert::ToDouble(textBox4->Text);
```

ეს იმას ნიშნავს, რომ ჩვენ მნიშვნელობები შეგვიძლია მივანიჭოთ მხოლოდ იმ სვეტებს, რომლებიც მოთავსებულია SELECT მოთხოვნაში ობიეტი-ადაპტერის შექმნისას. ზოგად შემთხვევაში, უნდა მივუთითოთ ის სვეტები, რომლებთანაც მუშაობას ვაპირებთ. თუ სვეტების სახელების ნაცვლად მივუთითებთ '*' სიმბოლოს, მაშინ აირჩევა ყველა სვეტი.

სტრიქნების წაშლა

ახლა ვნახოთ თუ როგორ ხდება ცხრილიდან სტრიქონის წაშლა. ამისთვის გამოიყენება DataRow ობიექტის Delete() მეთოდი. მოცემულ პროგრამაში ჯერ იძებნება წასაშლელი სტრიქონი, შემდეგ ის წაიშლება. Delete() მეთოდის შესრულების შემდეგ აუცილებელია Update() მეთოდის შესრულება. საქმე ის არის, რომ Delete() მეთოდი არ შლის სტრიქონს, არამედ

მონიშნავს მას შემდგომი წაშლისთვის. Update() მეთოდი რეალურად აფიქსირებს ცვლილებებს, ჩვენს შემთხვევაში, წასაშლელად მონიშნული სტრიქონის წაშლას.

```
{
//      პროგრამა 16.11
//      პროგრამით ხდება Personali ცხრილიდან სტრიქონის წაშლა
label1->Text = "";
//      find_key ცვლადში უნდა მოვათავსოთ საძებნი მნიშვნელობა
int find_key = Convert::ToInt32(textBox1->Text);
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT  personaliID, gvari, xelfasi, staji,
tarigi_dabadebis FROM Personali", myConnection);
//      SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//      პირველადი გასაღების ფორმირება
array<DataColumn^>^ keys = gcnew array<DataColumn^>(1);
keys[0] = myDataset->Tables["Personali"]->Columns["PersonaliID"];
myDataset->Tables["Personali"]->PrimaryKey = keys;
//      findRow ობიექტს ენიჭება ნაპოვნი სტრიქონი
DataRow^ findRow = myDataset->Tables["Personali"]->Rows->Find(find_key);
//      მოწმდება სტრიქონის არსებობა
if ( findRow->IsNull("PersonaliID") == false )
{
//      სტრიქონის წაშლა
findRow->Delete();
myAdapter->Update(myDataset, "Personali");
label1->Text = L"სტრიქონი წაიშალა";
}
else label1->Text = L"წასაშლელი სტრიქონი ვერ მოიძებნა";
//      შეერთების დახურვა
myConnection->Close();
}
```

რადგან, ხშირ შემთხვევაში, პირველადი გასაღებების ცოდნა შეუძლებელია, ამიტომ შეგვიძლია სტრიქონი მოვძებნოთ თანამშრომლის გვარის მიხედვით. ამისათვის, გამოვიყენებთ BindingSource ობიექტის Find() მეთოდს. მისი პირველი პარამეტრია იმ სვეტის სახელი, რომელშიც უნდა შესრულდეს ძებნა, მეორე პარამეტრია საძებნი სიდიდე. თუ საძებნი სიდიდე მოიძებნა, მაშინ გაიცემა ნაპოვნი სტრიქონის ინდექსი, წინააღმდეგ შემთხვევაში კი - -1. მოცემული პროგრამით ხდება Personali ცხრილიდან სტრიქონის წაშლის დემონსტრირება:

```
{
```

```
//      პროგრამა 16.12
//      პროგრამის მეშვეობით Personali ცხრილიდან წაშლება სტრიქონი
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT  personaliID, gvari, xelfasi, staji,
tarigi_dabadebis FROM Personali", myConnection);
//      SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//      bind ობიექტის დაკავშირება Personali ცხრილთან
BindingSource^ bind = gcnew BindingSource(myDataset, myDataset->Tables["Personali"]->ToString());
//      index ცვლადში იწერება მოძებნილი სტრიქონის ნომერი
int index = bind->Find("gvari", textBox1->Text);
label1->Text = index.ToString() + "\n";
//      პირველადი გასაღების ფორმირება
array<DataColumn^>^ keys = gcnew array<DataColumn^>(1);
keys[0] = myDataset->Tables["Personali"]->Columns["PersonaliID"];
myDataset->Tables["Personali"]->PrimaryKey = keys;
if ( index != -1 )
{
//      ნაპოვნი სტრიქონის მინიჭება findRow ობიექტისთვის
DataRow^ findRow = myDataset->Tables["Personali"]->Rows[index];
label1->Text += findRow["PersonaliID"]->ToString() + "   " + findRow["gvari"];
//      მოწმდება სტრიქონის არსებობა
if ( findRow->IsNull("PersonaliID") == false )
{
//      სტრიქონის წაშლა
findRow->Delete();
myAdapter->Update(myDataset, "Personali");
label1->Text += L"    სტრიქონი წაიშალა";
}
 else label1->Text += L"წასაშლელი სტრიქონი ვერ მოიძებნა";
}
//      შეერთების დახურვა
myConnection->Close();
}
```

ორ ცხრილს შორის ბმის შექმნა

ორ ცხრილს შორის ბმის (კავშირის, რელაციის) შესაქმნელად გამოიყენება DataRelation

ობიექტი. ის კავშირს ამყარებს DataTable ობიექტებს შორის. ბმა დავამყაროთ Personali და Shemkveti ცხრილებს შორის. Personali ცხრილში მოთავსებულია მონაცემები ფირმის თანამშრომლების შესახებ, Shemkveti ცხრილში კი - შემკვეთების შესახებ. თითოეულ თანამშრომელს შეიძლება ჰყავდეს ერთი ან მეტი შემკვეთი. მთავარია Personali ცხრილი. მისი პირველადი გასაღებია personaliID სვეტი. Shemkveti ცხრილი დამოკიდებულია. მისი გარე გასაღებია personaliID სვეტი. ამ ცხრილებს შორის კავშირის დასამყარებლად Personali ცხრილის პირველადი გასაღები უნდა დავუკავშიროთ Shemkveti ცხრილის გარე გასაღებს.

DataRelation ობიექტის შესაქმნელად გამოიყენება Relations ობიექტის Add() მეთოდი:

```
DataRelation^ personaliShemkvetiRelation = myDataset->Relations->Add("PersonaliShemkveti",
                                    myDataset->Tables["Personali"]->Columns["PersonaliID"],
                                    myDataset->Tables["Shemkveti"]->Columns["PersonaliID"]);
```

Add() მეთოდის პირველი პარამეტრი ბმის სახელია, მეორე - პირველადი გასაღები, მესამე კი - გარე გასაღები.

მოცემული პროგრამით ხდება ორ ცხრილს შორის კავშირის დამყარება:

```
{
//      პროგრამა 16.13
//      Personali და Shemkveti ცხრილებს შორის კავშირის დამყარება
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew SqlConnection("server=ROMANI-PC; database=Shekveta;
uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT  personaliID, gvari, xelfasi,
tarigi_dabadebis FROM Personali", myConnection);
//      SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      DataAdapter ობიექტების შექმნა თითოეული ცხრილისთვის და მათი შევსება
SqlDataAdapter^ personaliAdapter = gcnew SqlDataAdapter("SELECT * FROM Personali",
myConnection);
SqlDataAdapter^ shemkvetiAdapter = gcnew SqlDataAdapter("SELECT * FROM Shemkveti",
myConnection);
//      myDataset ობიექტის შევსება Personali და Shemkveti ცხრილებით
personaliAdapter->Fill(myDataset, "Personali");
shemkvetiAdapter->Fill(myDataset, "Shemkveti");
//      კავშირის დამყარება ცხრილებს შორის
DataRelation^ personaliShemkvetiRelation = myDataset->Relations->Add("PersonaliShemkveti",
myDataset->Tables["Personali"]->Columns["PersonaliID"],
myDataset->Tables["Shemkveti"]->Columns["PersonaliID"]);
//      ეკრანზე სტრიქონების გამოტანა
for each ( DataRow^ personaliRow in myDataset->Tables["Personali"]->Rows )
{
 label1->Text += personaliRow["PersonaliID"]->ToString() + "    " + personaliRow["gvari"] + "\n";
  for each ( DataRow^ shemkvetiRow in personaliRow->GetChildRows(personaliShemkvetiRelation) )
```

```
    label1->Text += "     " + shemkvetiRow["ShemkvetiID"]->ToString() + "  " + shemkvetiRow["gvari"] +
"\n";
}
//        შეერთების დახურვა
myConnection->Close();
}
```

როგორც პროგრამიდან ჩანს, ორივე ცხრილისთვის იქმნება შესაბამისი personaliAdapter და shemkvetiAdapter ადაპტერები. შემდეგ ხდება myDataset ობიექტის შევსება ორივე ცხრილით. ცხრილებს შორის ბმის დასამყარებლად იქმნება personaliShemkvetiRelation ობიექტი. გარე foreach ციკლს label კომპონენტში გამოაქვს Personali ცხრილის სტრიქონები, შიგა foreach ციკლს კი – Shemkveti ცხრილის სტრიქონები. იმისათვის, რომ Personali ცხრილის თითოეული სტრიქონისთვის გამოვიტანოთ შესაბამისი ბმული სტრიქონები Shemkveti ცხრილიდან, უნდა გამოვიყენოთ DataRow ობიექტის GetChildRows() მეთოდი, რომელსაც პარამეტრად უნდა გადავცეთ personaliShemkvetiRelation ობიექტი.

SQL ბრძანებების უშუალო შესრულება

თუ საჭიროა ოპერაციების შესრულება სტრიქონების ჯგუფზე, მაგალითად, სტრიქონების წაშლა ან სტრიქონებში სვეტების მნიშვნელობების შეცვლა, მაშინ დიდი ცხრილებისთვის გაცილებით ეფექტური იქნება ამის გაკეთება ერთი SQL-ბრძანებით. SQL-ბრძანების შესასრულებლად შეგვიძლია გამოვიყენოთ SqlComman და OleDbCommand კლასების ობიექტები. ამ ობიექტებს აქვს მეთოდები, რომლებიც SQL-ბრძანებებს უშუალოდ ასრულებს.

ერთი მნიშვნელობის მიღება

როდესაც საჭიროა SQL მოთხოვნისაგან ერთი შედეგის მიღება, მაგალითად პერსონალის საშუალო ასაკის, მაქსიმალური ხელფასის გაცემა და ა.შ. მაშინ უმჯობესია SlqCommand ობიექტის ExecuteScalar() მეთოდის გამოყენება. ის გასცემს სკალარულ მნიშვნელობას. მოცემული პროგრამის შესრულებით გაიცემა ფირმაში მომუშავე თანამშრომლების რაოდენობა:

```
{
//        პროგრამა 16.14
//        პროგრამით გაიცემა
label1->Text = "";
//        შეერთების ობიექტის შექმნა Personali ცხრილის სტრიქონების რაოდენობა
SqlConnection^ myConnection = gcnew
                SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        myCommand ობიექტის შექმნა მიმდინარე შეერთებისთვის
SqlCommand^ myCommand = myConnection->CreateCommand();
//        SELECT ბრძანების ფორმირება
myCommand->CommandText = "SELECT COUNT(*) FROM Personali";
//        მიღებული შედეგის მოთავსება countResult ობიექტში
Object^ countResult = myCommand->ExecuteScalar();
//        შედეგის ეკრანზე გამოტანა
label1->Text = countResult->ToString();
//        შეერთების დახურვა
myConnection->Close();
```

```
}
```

პროგრამაში იქმნება myCommand ობიექტი, რომლის CommandText თვისებას ვანიჭებთ შესასრულებელ SELECT მოთხოვნას. ამ მოთხოვნის შესასრულებლად უნდა გამოვიძახოთ myCommand ობიექტის ExecuteScalar() მეთოდი. მისი შესრულების შედეგად გაიცემა ფირმის თანამშრომლების რაოდენობა.

თუ გვაინტერესებს იმ თანამშრომლების, საშუალო ასაკი, რომლებიც სამედიცინო განყოფილებაში მუშაობენ, SELECT მოთხოვნა უნდა ჩავწეროთ შემდეგნაირად:

```
myCommand->CommandText =
                    "SELECT AVG(asaki) FROM Personali WHERE ganyofileba = N'სამედიცინო'";
```

UPDATE მოთხოვნის უშუალოდ შესრულება

როგორც აღვნიშნეთ INSERT, UPDATE და DELETE ბრძანებების შესრულების შედეგად ცხრილში მონაცემები იცვლება, მათ შესასრულებლად კი გამოიყენება SqlCommand ობიექტის ExecuteNonQuery() მეთოდი, რომელიც, აგრეთვე გასცემს ამ ბრძანებებით შეცვლილი სტრიქონების რაოდენობას. მოცემული პროგრამით ხდება ფირმის თანამშრომლების ხელფასის გაზრდა 10%-ით.

```
{
//      პროგრამა 16.15
//      პროგრამის მეშვეობით უშუალოდ სრულდება UPDATE ბრძანება
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew
            SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      myCommand ობიექტის შექმნა
SqlCommand^ myCommand = myConnection->CreateCommand();
//      UPDATE ბრძანების მომზადება შესასრულებლად
myCommand->CommandText = "UPDATE Personali SET xelfasi = xelfasi * 1.1";
//      UPDATE ბრძანების უშუალოდ შესრულება
int rowsAffected = myCommand->ExecuteNonQuery();
//      დამუშავებული სტრიქონების რაოდენობს ეკრანზე გამოტანა
label1->Text = rowsAffected.ToString();
//      შეერთების დახურვა
myConnection->Close();
}
```

პროგრამაში იქმნება myCommand ობიექტი, რომლის CommandText თვისებას ენიჭება შესასრულებელი UPDATE ბრძანება. ამ ბრძანებას ასრულებს ამ ობიექტის ExecuteNonQuery() მეთოდი, რომელიც გასცემს შეცვლილი სტრიქონების რაოდენობას. ეს რაოდენობა მიენიჭება rowsAffected ცვლადს. ცვლილებების სანახავად ფორმაზე მოვათავსოთ Button კომპონენტი და მას მივაბათ **16.3 პროგრამა**.

თუ UPDATE ბრძანებას დავუმატებთ WHERE განყოფილებას, მაშინ შევძლებთ რამდენიმე სტრიქონში მონაცემების შეცვლას. მაგალითად, სამედიცინო განყოფილების პერსონალისთვის ხელფასის 10%-ით გასაზრდელად UPDATE ბრძანება ასე უნდა შევიტანოთ:

```
myCommand->CommandText =
                    "UPDATE Personali SET xelfasi = xelfasi * 1.1 WHERE ganyofileba = N'სამედიცინო'";
```

ახლა განვიხილოთ შემთხვევა, როდესაც UPDATE ბრძანებას გადაეცემა პარამეტრები.

მოცემული პროგრამით ხდება Shekveta მონაცემთა ბაზის Personali ცხრილის მონაცემების შეცვლა იმ სტრიქონში, რომლის პირველადი გასაღების მნიშვნელობა დაემთხვევა პარამეტრად გადაცემულ მნიშვნელობას:

```
{
//      პროგრამა 16.16
//      პროგრამაში ხდება UPDATE მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection = gcnew
SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      UPDATE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText = "UPDATE Personali SET gvari = @gvari, ganyofileba =
@ganyofileba, qalaqi = @qalaqi, " + "xelfasi = @xelfasi, asaki = @asaki, staji = @staji, tarigi_dabadebis =
@tarigi_dabadebis " + "WHERE (personaliID = @personaliID)";
//      პარამეტრების დამატება
mySqlCommand->Parameters->Add("@personaliID", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@gvari", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@qalaqi", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@xelfasi", SqlDbType::Float, 8);
mySqlCommand->Parameters->Add("@asaki", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@staji", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@tarigi_dabadebis", SqlDbType::DateTime, 8);
//      პარამეტრებისთვის მნიშვნელობების მინიჭება
mySqlCommand->Parameters["@personaliID"]->Value = Convert::ToInt32(textBox7->Text);
mySqlCommand->Parameters["@gvari"]->Value = textBox1->Text;
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox2->Text;
mySqlCommand->Parameters["@qalaqi"]->Value = textBox3->Text;
mySqlCommand->Parameters["@xelfasi"]->Value = Convert::ToDouble(textBox4->Text);
mySqlCommand->Parameters["@asaki"]->Value = Convert::ToInt32(textBox5->Text);
mySqlCommand->Parameters["@staji"]->Value = Convert::ToInt32(textBox6->Text);
mySqlCommand->Parameters["@tarigi_dabadebis"]->Value = dateTimePicker1->Value;
//      UPDATE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
//      დამუშავებული სტრიქონების რაოდენობის გამოტანა ეკრანზე
label1->Text = rowsAffected.ToString();
//      შეერთების დახურვა
myConnection->Close();
}
```

როგორც ვხედავთ, პარამეტრის დამატებისას პარამეტრის სახელის გარდა მითითებულია მისი ტიპიც (ცხრილი 16.1).

შესაცვლელი სტრიქონი შეგვიძლია მოვძებნოთ bindingSource1 კომპონენტის Find() ბრძანების საშუალებით და გამოვიყენოთ ნაპოვნი სტრიქონის პირველადი გასაღები. პროგრამას ექნება შემდეგი სახე:

```
{
//      პროგრამა 16.17
//      პროგრამაში ხდება UPDATE მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      ადაპტერის შექმნა
SqlDataAdapter^ myAdapter = gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//      SqlCommandBuilder ობიექტის შექმნა SQL ბრძანების ასაგებად
SqlCommandBuilder^ myBuilder = gcnew SqlCommandBuilder(myAdapter);
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      myDataset ობიექტის შევსება Personali ცხრილის სტრიქონებით
myAdapter->Fill(myDataset, "Personali");
//      personaliBind ობიექტის დაკავშირება Personali ცხრილთან
BindingSource^ personaliBind = gcnew BindingSource(myDataset, "Personali");
//      ნაპოვნი სტრიქონის ინდექსის მიღება
int index = personaliBind->Find("gvari", textBox1->Text);
if ( index != -1 )
{
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      UPDATE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
"UPDATE Personali SET ganyofileba = @ganyofileba, qalaqi = @qalaqi, " +
        "xelfasi = @xelfasi, asaki = @asaki, staji = @staji, tarigi_dabadebis = @tarigi_dabadebis " +
        "WHERE (personaliID = @personaliID)";
//      პარამეტრების დამატება
mySqlCommand->Parameters->Add("@personaliID", SqlDbType::Int, 4);
//mySqlCommand->Parameters->Add("@gvari", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@qalaqi", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@xelfasi", SqlDbType::Float, 8);
mySqlCommand->Parameters->Add("@asaki", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@staji", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@tarigi_dabadebis", SqlDbType::DateTime, 8);
//      პარამეტრებისთვის მნიშვნელობების მინიჭება
mySqlCommand->Parameters["@personaliID"]->Value =
                                    Convert::ToInt32(myDataset->Tables["Personali"]->Rows[index][0]);
//mySqlCommand->Parameters["@gvari"]->Value = textBox1->Text;
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox2->Text;
mySqlCommand->Parameters["@qalaqi"]->Value = textBox3->Text;
mySqlCommand->Parameters["@xelfasi"]->Value = Convert::ToDouble(textBox4->Text);
mySqlCommand->Parameters["@asaki"]->Value = Convert::ToInt32(textBox5->Text);
mySqlCommand->Parameters["@staji"]->Value = Convert::ToInt32(textBox6->Text);
```

```
mySqlCommand->Parameters["@tarigi_dabadebis"]->Value = dateTimePicker1->Value;
//        UPDATE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
}
else label1->Text = L"სტრიქონი არ არსებობს";
//        შეერთების დახურვა
myConnection->Close();
}
```

შესაძლებელია, აგრეთვე რამდენიმე სტრიქონის შეცვლა. მოცემულ პროგრამაში ხდება მითითებული განყოფილების თანამშრომლების ხელფასის გაზრდა მითითებული თანხით. განყოფილებისა და ხელფასის ნაზრდის ზომა შეგვიძლია textBox კომპონენტებიდან შევიტანოთ:

```
{
//        პროგრამა 16.18
//        პროგრამის მეშვეობით ხდება UPDATE მოთხოვნის შესრულების დემონსტრირება
//        შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
myConnection->Open();
//        mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//        UPDATE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
        "UPDATE Personali SET xelfasi = xelfasi + @xelfasi WHERE (ganyofileba = @ganyofileba)";
//        პარამეტრების დამატება
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@xelfasi", SqlDbType::Float, 8);
//        პარამეტრებისთვის მნიშვნელობების მინიჭება
mySqlCommand->Parameters["@xelfasi"]->Value = Convert::ToDouble(textBox1->Text);
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox1->Text;
//        UPDATE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
//        შეერთების დახურვა
myConnection->Close();
}
```

ცხრილი 16.1. SqlDbType ჩამოთვლადი ტიპის ელემენტები

ელემენტი	აღწერა
BigInt	64-ბიტიანი ნიშნიანი მთელი რიცხვი, რომელიც იღებს მნიშვნელობას -263÷263-1 (-9223372036854775808÷9223372036854775807)
Binary	ბაიტების მასივი, რომელშიც 8000 ელემენტამდეა
Bit	უნიშნო მთელი რიცხვი, რომელიც იღებს მნიშვნელობებს 0, 1 ან null
Char	სტრიქონი, რომელიც შეიცავს 8000 სიმბოლომდე და არ შეიცავს Unicode სიმბოლოებს
DateTime	დრო და თარიღი 1753 წლის 1 იანვრიდან 9999 წლის 31 დეკემბრამდე
Decimal	მითითებული სიზუსტის და მასშტაბის რიცხვი $-10^{38}-1 \div 10^{38}-1$ (-79228162514264337593543950335÷ 79228162514264337593543950335)
Float	მოძრავწერტილიანი რიცხვი დიაპაზონში -1.79E+308÷1.79E+308
Image	ბაიტების მასივი, რომელშიც $2^{31}-1$ (2147483647) ელემენტამდეა
Int	32-თანრიგა მთელი რიცხვი ნიშნით, დიაპაზონში $-2^{31} \div 2^{31}-1$ (-2147483648÷2147483647)
Money	ფული დიაპაზონში -922337203685477.5808÷922337203685477.5807
NChar	Unicode სიმბოლოების სტრიქონი. მაქსიმალური სიგრძეა 4000 სიმბოლო
NText	Unicode სიმბოლოების სტრიქონი. მაქსიმალური სიგრძეა $2^{30}-1$ (1073741823) სიმბოლო
NVarChar	Unicode სიმბოლოების სტრიქონი. მაქსიმალური სიგრძეა 4000 სიმბოლო
Real	მოძრავწერტილიანი რიცხვი დიაპაზონში -3.40E+38÷3.40E+38
SmallDateTime	დრო და თარიღი 1900 წლის 1 იანვრიდან 2079 წლის 6 ივნისამდე
SmallInt	16-ბიტიანი ნიშნიანი მთელი რიცხვი
SmallMoney	ფული დიაპაზონში -214748.3648÷214748.3647
Text	სტრიქონი, რომელშიც $2^{31}-1$ (2147483647) სიმბოლოა და არ შეიცავს Unicode სიმბოლოებს
Timestamp	თარიღი და დრო yyyymmddhhmmss ფორმატში (წელი/თვე/დღე/საათი/წუთები/წამები)
TinyInt	8-ბიტიანი უნიშნო მთელი რიცხვი დიაპაზონში $0 \div 2^8-1$ (255)
UniqueIdentifier	GUID (Globally Unique IDentifier - გლობალური უნიკალური იდენტიფიკატორი)
VarBinary	ბაიტების მასივი, რომელშიც 8000 ელემენტამდეა
VarChar	სტრიქონი მაქსიმალური სიგრძით 8000 სიმბოლომდე, რომელიც არ შეიცავს Unicode სიმბოლოებს
Variant	მონაცემების ტიპი, რომელიც შეიძლება შეიცავდეს რიცხვებს, სტრიქონებს, ბაიტებსა და თარიღებს

INSERT მოთხოვნის უშუალოდ შესრულება

Shekveta მონაცემთა ბაზის Personali ცხრილს შეგვიძლია დავუმატოთ ახალი სტრიქონი INSERT მოთხოვნის უშუალოდ შესრულების გზით. ამის დემონსტრირება ხდება მოცემული პროგრამით:

```
{
//      პროგრამა 16.19
//      პროგრამის მეშვეობით ხდება INSERT მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
```

```
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      INSERT ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
        "INSERT INTO Personali (gvari, ganyofileba, qalaqi, xelfasi, asaki, staji, tarigi_dabadebis) " +
        "VALUES (@gvari, @ganyofileba, @qalaqi, @xelfasi, @asaki, @staji, @tarigi_dabadebis)";
//      პარამეტრების დამატება
mySqlCommand->Parameters->Add("@gvari", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@qalaqi", SqlDbType::NVarChar, 20);
mySqlCommand->Parameters->Add("@xelfasi", SqlDbType::Float, 8);
mySqlCommand->Parameters->Add("@asaki", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@staji", SqlDbType::Int, 4);
mySqlCommand->Parameters->Add("@tarigi_dabadebis", SqlDbType::DateTime, 8);
//      პარამეტრებისთვის მნიშვნელობების მინიჭება
mySqlCommand->Parameters["@gvari"]->Value = textBox1->Text;
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox2->Text;
mySqlCommand->Parameters["@qalaqi"]->Value = textBox3->Text;
mySqlCommand->Parameters["@xelfasi"]->Value = Convert::ToDouble(textBox4->Text);
mySqlCommand->Parameters["@asaki"]->Value = Convert::ToInt32(textBox5->Text);
mySqlCommand->Parameters["@staji"]->Value = Convert::ToInt32(textBox6->Text);
mySqlCommand->Parameters["@tarigi_dabadebis"]->Value = dateTimePicker1->Value;
//      INSERT ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
//      შეერთების დახურვა
myConnection->Close();
}
```

როგორც ვხედავთ, INSERT მოთხოვნაში არ არის მითითებული Personali ცხრილის პირველადი გასაღები personaliID, ვინაიდან ამ სვეტისთვის Identity თვისება დაყენებულია Yes მდგომარეობაში და სტრიქონის დამატებისას ავტომატურად ხდება პირველადი გასაღებისთვის სწორი მნიშვნელობის გენერირება.

DELETE მოთხოვნის უშუალოდ შესრულება

Shekveta მონაცემთა ბაზის Personali ცხრილიდან წავშალოთ ერთი სტრიქონი DELETE მოთხოვნის უშუალოდ შესრულების გზით. წასაშლელი სტრიქონის პირველადი გასაღების მნიშვნელობა შეგვაქვს textBox კომპონენტიდან. ამის დემონსტრირება ხდება მოცემული პროგრამით:

```
{
//      პროგრამა 16.20
//      პროგრამის მეშვეობით ხდება DELETE მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
```

```
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      DELETE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText = "DELETE FROM Personali WHERE (PersonaliID = @PersonaliID)";
//      mySqlCommand ობიექტისთვის პარამეტრების დამატება
mySqlCommand->Parameters->Add("@PersonaliID", SqlDbType::Int, 4);
mySqlCommand->Parameters["@PersonaliID"]->Value = Convert::ToInt32(textBox1->Text);
//      DELETE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
//      შეერთების დახურვა
myConnection->Close();
}
```

წასაშლელი სტრიქონი შეგვიძლია მოვძებნოთ bindingSource1 კომპონენტის Find() ბრძანების საშუალებით და გამოვიყენოთ ნაპოვნი სტრიქონის პირველადი გასაღები. დავუშვათ, Personali ცხრილიდან გვინდა წავშალოთ მითითებული გვარის შემცველი სტრიქონი. ამისათვის, ძებნა უნდა შევასრულოთ "gvari" სვეტში, საძებნი გვარი კი შევიტანოთ textBox1 კომპონენტში. ამის დემონსტრირება ხდება მოცემული პროგრამით:

```
{
//      პროგრამა 16.21
//      პროგრამით ხდება DELETE მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      ადაპტერის შექმნა და დაკავშირება SELECT მოთხოვნასთან
SqlDataAdapter^ personaliAdapter =
                                gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//      myDataset ობიექტის შევსება Personali ცხრილით
personaliAdapter->Fill(myDataset, "Personali");
//      bind1 ობიექტის შექმნა და დაკავშირება myDataset ობიექტის Personali ცხრილთან
BindingSource^ bind1 = gcnew BindingSource(myDataset, "Personali");
//      ნაპოვნი სტრიქონის ინდექსის მიღება
int index = bind1->Find("gvari", textBox1->Text);
if ( index != -1 )
{
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      DELETE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText = "DELETE FROM Personali WHERE (PersonaliID = @PersonaliID)";
//      mySqlCommand ობიექტისთვის პარამეტრის დამატება
mySqlCommand->Parameters->Add("@PersonaliID", SqlDbType::Int, 4);
```

```
mySqlCommand->Parameters["@PersonaliID"]->Value =
                                    Convert::ToInt32(myDataset->Tables["Personali"]->Rows[index][0]);
//      DELETE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
}
//      შეერთების დახურვა
myConnection->Close();
}
```

შესაძლებელია, აგრეთვე, რამდენიმე სტრიქონის წაშლა. მოცემულ პროგრამაში ხდება მონაცემების (სტრიქონების) წაშლა იმ თანამშრომლების შესახებ, რომლებიც დაბადებული არიან მითითებულ თარიღზე ადრე. თარიღის შესატანად შეგვიძლია dateTimePicker კომპონენტის გამოყენება.

```
{
//      პროგრამა 16.22
//      პროგრამის მეშვეობით ხდება DELETE მოთხოვნის შესრულების დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      DataSet ობიექტის შექმნა ცხრილების, სტრიქონებისა და სვეტების შესანახად
DataSet^ myDataset = gcnew DataSet();
//      ადაპტერის შექმნა და მისთვის SELECT მოთხოვნის დაკავშირება
SqlDataAdapter^ personaliAdapter =
                                    gcnew SqlDataAdapter("SELECT * FROM Personali", myConnection);
//      myDataset ობიექტის შევსება Personali ცხრილით
personaliAdapter->Fill(myDataset, "Personali");
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      DELETE ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
                        "DELETE FROM Personali WHERE (tarigi_dabadebis < @tarigi_dabadebis)";
//      mySqlCommand ობიექტისთვის პარამეტრის დამატება
mySqlCommand->Parameters->Add("@tarigi_dabadebis", SqlDbType::DateTime, 8);
mySqlCommand->Parameters["@tarigi_dabadebis"]->Value = dateTimePicker1->Value;
//      DELETE ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString();
//      შეერთების დახურვა
myConnection->Close();
}
```

ტრანზაქციებთან მუშაობა ADO.NET საშუალებებით

ADO.NET გარემოში ტრანზაქციებთან სამუშაოდ SqlTransaction ობიექტი გამოიყენება. დავუშვათ გვინდა, რომ Personali და Shemkveti ცხრილებს INSERT ბრძანების გამოყენებით თითო სტრიქონი დავუმატოთ და ეს მოქმედებები ერთ ტრანზაქციაში გავაერთიანოთ.

ამისათვის, უნდა შევასრულოთ შემდეგი მოქმედებები:
1. SqlTransaction ობიექტის შექმნა და ტრანზაქციის დაწყება SqlConnection ობიექტის BeginTransaction() მეთოდის გამოძახების გზით.
2. SqlCommand ობიექტის შექმნა SQL მოთხოვნის შესანახად.
3. SqlCommand ობიექტის Transaction თვისების დაყენება.
4. პირველი INSERT მოთხოვნის შემცველი სტრიქონის ფორმირება.
5. SqlCommand ობიექტის CommandText თვისებისათვის პირველი INSERT მოთხოვნის შემცველი სტრიქონის მინიჭება.
6. პირველი INSERT მოთხოვნის შესრულება SqlCommand ობიექტის ExecuteNonQuery() მეთოდის საშუალებით.
7. მეორე INSERT მოთხოვნის შემცველი სტრიქონის ფორმირება.
8. SqlCommand ობიექტის CommandText თვისებისათვის მეორე INSERT მოთხოვნის შემცველი სტრიქონის მინიჭება.
9. მეორე INSERT მოთხოვნის შესრულება SqlCommand ობიექტის ExecuteNonQuery() მეთოდის საშუალებით.
10. ტრანზაქციის დაფიქსირება SqlTransaction ობიექტის Commit() მეთოდის გამოყენებით.

ქვემოთ მოყვანილია პროგრამა, რომელშიც აღწერილი მოქმედებები სრულდება:

```
{
//      პროგრამა 16.23
//      პროგრამაში ხდება ტრანზაქციასთან მუშაობის დემონსტრირება
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
        gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      mySqlTransaction ობიექტის შექმნა ტრანზაქციის დასაწყებად
SqlTransaction^ mySqlTransaction = myConnection->BeginTransaction();
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = myConnection->CreateCommand();
//      mySqlTransaction ობიექტის დაკავშირება mySqlCommand ობიექტთან
mySqlCommand->Transaction = mySqlTransaction;
//      INSERT ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
        L"INSERT INTO Personali (gvari, ganyofileba, qalaqi, xelfasi, asaki, staji, tarigi_dabadebis) " +
        L"VALUES (N'კირვალიძე ნინო', N'სასპორტო', N'ონი', 555.55, 50, 5, '2005.11.23')";
//      INSERT ბრძანების შესრულება
int rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text = rowsAffected.ToString() + "    ";
//      INSERT ბრძანების მომზადება შესასრულებლად
mySqlCommand->CommandText =
        L"INSERT INTO Personali (gvari, ganyofileba, qalaqi, xelfasi, asaki, staji, tarigi_dabadebis) " +
        L"VALUES (N'ჭუმბურიძე ნინო', N'სასპორტო', N'ბოსლევი', 777.55, 70, 7, '2000.05.20')";
//      INSERT ბრძანების შესრულება
rowsAffected = mySqlCommand->ExecuteNonQuery();
label1->Text += rowsAffected.ToString();
//      ტრანზაქციის დაფიქსირება
mySqlTransaction->Commit();
```

```
//      შეერთების დახურვა
myConnection->Close();
}
```

შენახული პროცედურის გამოძახება

SQL-ბრძანებების შესრულების გარდა, შეგვიძლია შესასრულებლად გამოვიძახოთ შენახული პროცედურა და ფუნქცია და საჭიროების შემთხვევაში გადავცეთ პარამეტრები. თუ შენახული პროცედურა გასცემს სკალარულ შედეგს, მაშინ პროცედურის შესასრულებლად უნდა გამოვიყენოთ ExecuteScalar() და ExecuteNonQuery() მეთოდები. თუ პროცედურა გასცემს სტრიქონებს, მაშინ მის შესასრულებლად უნდა გამოვიყენოთ ExecuteReader() მეთოდი.

დავუშვათ, გვინდა შევასრულოთ შენახული პროცედურა, რომელიც გასცემს მითითებული განყოფილების თანამშრომლების შესახებ მონაცემებს. მოცემული პროგრამის მეშვეობით გამოიძახება ეს შენახული პროცედურა და მას ერთი სტრიქონული ტიპის პარამეტრი გადაეცემა, რომელიც განყოფილების სახელს შეიცავს:

```
{
//      პროგრამა 16.24
//      პროგრამის მეშვეობით ხდება შენახული პროცედურის შესრულების დემონსტრირება
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
myConnection->Open();
//      myCommand ობიექტის შექმნა
SqlCommand^ myCommand = myConnection->CreateCommand();
//      myCommand ობიექტის დაკავშირება Myproc_Par პროცედურასთან
myCommand->CommandType = CommandType::StoredProcedure;
myCommand->CommandText = "Myproc_Par";
//      myCommand ობიექტისთვის პარამეტრის დამატება
myCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 30);
myCommand->Parameters["@ganyofileba"]->Value = textBox1->Text;
//      Myproc_Par შენახული პროცედურის შესრულება და
//      მიღებული სტრიქონების ჩაწერა myReader ობიექტში
SqlDataReader^ myReader = myCommand->ExecuteReader();
//      myReader ობიექტიდან სტრიქონების გამოტანა ეკრანზე
for ( ; myReader->Read(); )
        label1->Text += myReader["gvari"]->ToString() + "   " + myReader["qalaqi"] + "   " +
                        myReader["staji"]->ToString() + "\n";
//      წამკითხველის დახურვა
myReader->Close();
//      შეერთების დახურვა
myConnection->Close();
}
```

როგორც პროგრამიდან ჩანს, Myproc_Par პროცედურის სახელი უნდა მივუთითოთ myCommand ობიექტის CommandText თვისებაში. რაც შეეხება პარამეტრს, ის უნდა დავუმატოთ ამავე ობიექტის პარამეტრების კოლექციას Add() მეთოდის გამოყენებით:

```
myCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 30);
```

შემდეგ, ამ პარამეტრს უნდა მივანიჭოთ მნიშვნელობა უშუალოდ ან textBox

კომპონენტიდან. ამის შემდეგ, შეგვიძლია შევასრულოთ myCommand ობიექტის ExecuteReader() მეთოდი.

შენახული პროცედურის შესრულება შეიძლება, აგრეთვე ადაპტერის Fill() მეთოდის გამოყენებით. ამ პროგრამის მეშვეობით ხდება შენახული პროცედურის შესრულების დემონსტრირება, რომელსაც ერთი პარამეტრი აქვს. პროცედურას ეკრანზე გამოაქვს მონაცემები მითითებული განყოფილების თანამშრომლების შესახებ:

```
{
//      პროგრამა 16.25
//      პროგრამაში ხდება Myproc_Par შენახული პროცედურის გამოძახება და მისთვის
//      პარამეტრის გადაცემა
//      შეერთების შექმნა
SqlConnection^ mySqlConnection =
        gcnew SqlConnection("server=ROMANI-PC;database=Shekveta;uid=sa;pwd=paroli");
//      შეერთების გახსნა
mySqlConnection->Open();
//      mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = mySqlConnection->CreateCommand();
//      mySqlCommand ობიექტისთვის პარამეტრების დამატება
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 30);
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox1->Text;
//      mySqlCommand ობიექტის მომზადება შენახული პროცედურის შესასრულებლად
mySqlCommand->CommandText = "dbo.Myproc_Par";
mySqlCommand->CommandType = CommandType::StoredProcedure;
//      ადაპტერის შექმნა და მომზადება სამუშაოდ
SqlDataAdapter^ mySqlDataAdapter = gcnew SqlDataAdapter();
mySqlDataAdapter->SelectCommand = mySqlCommand;
//      myDataSet ობიექტის შექმნა
DataSet^ myDataSet = gcnew DataSet();
//      Myproc_Par შენახული პროცედურის შესრულება
mySqlDataAdapter->Fill(myDataSet, "Myproc_Par");
//      შედეგის ეკრანზე გამოტანა
dataGridView1->DataSource = myDataSet;
dataGridView1->DataMember = "Myproc_Par";
//      შეერთების დახურვა
mySqlConnection->Close();
}
```

გამოვიძახოთ შენახული პროცედურა, რომელიც გასცემს მითითებული განყოფილების თანამშრომლების რაოდენობას. ამისათვის გამოვიყენოთ SqlCommand ობიექტის ExecuteScalar() მეთოდი. მოყვანილი პროგრამით ხდება ამის დემონსტრირება:

```
{
//      პროგრამა 16.26
//      პროგრამით ხდება პროცედურის გამოძახების დემონსტრირება ExecuteScalar() მეთოდით
label1->Text = "";
//      შეერთების ობიექტის შექმნა
SqlConnection^ myConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//      შეერთების გახსნა
```

```
myConnection->Open();
//        myCommand ობიექტის შექმნა
SqlCommand^ myCommand = myConnection->CreateCommand();
//        ბრძანების ტიპის არჩევა
myCommand->CommandType = CommandType::StoredProcedure;
myCommand->CommandText = "Myproc_Par";
//        myCommand ობიექტისთვის პარამეტრის დამატება
myCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 30);
myCommand->Parameters["@ganyofileba"]->Value = textBox1->Text;
//        Myproc_Par შენახული პროცედურის შესრულება
Object^ countResult = myCommand->ExecuteScalar();
//        შედეგის ეკრანზე გამოტანა
label1->Text = countResult->ToString();
//        შეერთების დახურვა
myConnection->Close();
}
```

ფუნქციის გამოძახება

შენახული პროცედურის შესრულების გარდა, შეგვიძლია შესასრულებლად გამოვიძახოთ ფუნქცია და მას პარამეტრები გადავცეთ. შევასრულოთ ფუნქცია, რომელიც გასცემს მითითებული განყოფილების თანამშრომლების საშუალო ხელფასს. ამის დემონსტრირება ხდება შემდეგი პროგრამით:

```
{
//        პროგრამა 16.27
//        პროგრამით ხდება ფუნქციის გამოძახებისა და მისთვის
//        პარამეტრის გადაცემის დემონსტრირება
SqlConnection^ mySqlConnection =
                gcnew SqlConnection("server=ROMANI-PC; database=Shekveta; uid=sa; pwd=paroli");
//        შეერთების გახსნა
mySqlConnection->Open();
//        mySqlCommand ობიექტის შექმნა
SqlCommand^ mySqlCommand = mySqlConnection->CreateCommand();
//        mySqlCommand ობიექტისთვის პარამეტრის დამატება
mySqlCommand->Parameters->Add("@ganyofileba", SqlDbType::NVarChar, 30);
mySqlCommand->Parameters["@ganyofileba"]->Value = textBox1->Text;
//        SELECT მოთხოვნის ფორმირება
mySqlCommand->CommandText = "SELECT dbo.Maqsimaluri_Xelfasi(@ganyofileba)";
//        ადაპტერის ფორმირება
SqlDataAdapter^ mySqlDataAdapter = gcnew SqlDataAdapter();
//        SELECT ბრძანების მომზადება შესასრულებლად
mySqlDataAdapter->SelectCommand = mySqlCommand;
//        Maqsimaluri_Xelfasi ფუნქციის შესრულება
DataSet^ myDataSet = gcnew DataSet();
//        Maqsimaluri_Xelfasi ფუნქციის შესრულება
Object^ maqsimaluri_xelfasi = mySqlCommand->ExecuteScalar();
//        შედეგის ეკრანზე გამოტანა
label1->Text = maqsimaluri_xelfasi->ToString();
```

```
//          შეერთების დახურვა
mySqlConnection->Close();
}
```

დანართი 1. სავარჯიშოები თავების მიხედვით

თავი 2. C++ ენის საფუძვლები

არითმეტიკული გამოსახულება

შეადგინეთ პროგრამა, რომელიც გამოთვლის ქვემოთ მოცემული არითმეტიკული გამოსახულებების მნიშვნელობებს:

2.1.1. $$y = 1 + x + \frac{x^2}{2}$$

2.1.2. $$y = \frac{x^2 + z^2}{1 - \frac{x^2 - z^2}{2}}$$

2.1.3. $$y = \sqrt{1 + \sqrt{|x|}}$$

2.1.4. $$y = (1 + x)\frac{x - \frac{\sqrt{x-1}}{4}}{e^x + \frac{1}{x^2 + 4}}$$

2.1.5. $$y = \sin x^3 + \cos^3 x^4 - e^{\sqrt{x}}$$

შეადგინეთ პროგრამა, რომელიც გამოთვლის:

2.1.6. დენის ძალას, $J = \frac{U}{R}$.

2.1.7. ხუთი რიცხვის საშუალო არითმეტიკულს.

2.1.8. ოთხი რიცხვის საშუალო გეომეტრიულს.

2.1.9. სამკუთხედის ფართობს, $s = \frac{ah}{2}$.

2.1.10. სამკუთხედის ფართობს ჰერონის ფორმულის გამოყენებით $S = \sqrt{p(p-a)(p-b)(p-c)}$.

2.1.11. სამკუთხედის პერიმეტრს, $p = a + b + c$.

2.1.12. კვადრატის ფართობს.

2.1.13. მართკუთხედის პერიმეტრს.

2.1.14. წრეწირის სიგრძეს, $C = 2\pi\pi$.

2.1.15. წრეწირის ფართობს, $S = \pi r^2$.

2.1.16. ტრაპეციის ფართობს, $S = \frac{a+b}{2}h$.

2.1.17. ტრაპეციის პერიმეტრს.

ლოგიკური გამოსახულება

შეადგინეთ პროგრამა, რომელიც გამოთვლის ქვემოთ მოცემული ლოგიკური გამოსახულებების მნიშვნელობებს:

2.2.1. $y = x1 \&\& x2 \| x3 \&\& !x4$, სადაც x1=true, x2=false, x3=true, x4=false

2.2.2. $y = x1 \| !x2 \&\& x3 \| x4$, სადაც x1=false, x2=false, x3=true, x4=true

2.2.3. $y = !x1 \&\& !x2 \&\& x3 \| x4$, სადაც x1=true, x2=true, x3=false, x4=false

2.2.4. $y = !x1 \| x2 \| !x3 \&\& x4$, სადაც x1=false, x2=true, x3=true, x4=false

თავი 3. მმართველი ოპერატორი

if ოპერატორი

შეადგინეთ პროგრამა, რომელიც დაადგენს:

3.1.1. რიცხვი დადებითია თუ უარყოფითი. რიცხვი textBox კომპონენტიდან შეიტანეთ. label კომპონენტში გამოიტანეთ შესაბამისი შეტყობინება.
3.1.2. რიცხვი კენტია თუ ლუწი.
3.1.3. რიცხვი არის თუ არა 5-ის ჯერადი.
3.1.4. რიცხვი არის თუ არა 30-ის ტოლი.
3.1.5. რიცხვი მეტია თუ არა 10-ზე.
3.1.6. რიცხვი ნაკლებია თუ არა -2,01-ზე.
3.1.7. რიცხვი არის თუ არა მეტი ან ტოლი 15-ზე.
3.1.8. რიცხვი არის თუ არა ნაკლები ან ტოლი 23-ზე.
3.1.9. ორ რიცხვს შორის მაქსიმალურს.
3.1.10. ორ რიცხვს შორის მინიმალურს.

ჩადგმული if ოპერატორი

შეადგინეთ პროგრამა, რომელიც დაადგენს:

3.2.1. სამ რიცხვს შორის მაქსიმალურს.
3.2.2. სამ რიცხვს შორის მინიმალურს.
3.2.3. ორ რიცხვს შორის არის თუ არა დადებითი.
3.2.4. ორ რიცხვს შორის არის თუ არა უარყოფითი.
3.2.5. ორ რიცხვს შორის არის თუ არა 7-ის ჯერადი.
3.2.6. სამ რიცხვს შორის არის თუ არა ლუწი.
3.2.7. სამ რიცხვს შორის არის თუ არა დადებითი.
3.2.8. სამ რიცხვს შორის არის თუ არა უარყოფითი.
3.2.9. სამ რიცხვს შორის რამდენია კენტი.
3.2.10. სამ რიცხვს შორის რამდენია 5-ის ტოლი.
3.2.11. სამ რიცხვს შორის რამდენია 10-ზე მეტი.
3.2.12. ორ რიცხვს შორის რომელია 5-ის ჯერადი.
3.2.13. სამ რიცხვს შორის რომელია 50-ზე ნაკლები.

ლოგიკა

შეადგინეთ პროგრამა, რომელიც დაადგენს:

3.3.1. x რიცხვი არის თუ არა $0 < x \leq 17$ დიაპაზონში მოთავსებული.
3.3.2. x რიცხვი არის თუ არა $10 \leq x < 19$ დიაპაზონში მოთავსებული.

3.3.3. x რიცხვი არის თუ არა 25-ზე ნაკლები ან 100-ზე მეტი.
3.3.4. x რიცხვი არის თუ არა 30-ის ტოლი ან 5-ზე ნაკლები.
3.3.5. ორივე რიცხვი არის თუ არა დადებითი.
3.3.6. სამივე რიცხვი არის თუ არა 7-ის ჯერადი.

for, while, do-while ოპერატორები

შეადგინეთ პროგრამა, რომელიც გამოთვლის შემდეგი გამოსახულებების მნიშვნელობებს:

3.4.1. $\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5}$

3.4.2. $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9}$

3.4.3. $\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2}$

3.4.4. $\frac{\pi^2}{8} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \frac{1}{9^2}$

3.4.5. $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5}$

3.4.6. $\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5}$

3.4.7. $\mathrm{arctg}x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}$

3.4.8. $\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - x^5$

3.4.9. $\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + x^5$

3.4.10. $\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3$

3.4.11. $\frac{1}{\sqrt{1+x}} = 1 - \frac{1}{2}x + \frac{3}{8}x^2 - \frac{5}{16}x^3$

3.4.12. $\arcsin x = x + \frac{1}{2 \cdot 3}x^3 + \frac{1 \cdot 3}{3 \cdot 4 \cdot 5}x^5$

3.4.13. $\ln\frac{1+x}{1-x} = 2(x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7})$

3.4.14. $y = \frac{1 \cdot 2}{3 \cdot 4} + \frac{3 \cdot 4}{5 \cdot 6} + \frac{5 \cdot 6}{7 \cdot 8}$

შეადგინეთ პროგრამა, რომელიც:

3.4.15. დათვლის ლუწი რიცხვების რაოდენობას 5-დან 27-მდე დიაპაზონში.

3.4.16. შეკრებს კენტ რიცხვებს 10-დან 20-მდე დიაპაზონში.

3.4.17. დათვლის 8-ის ჯერადი რიცხვების რაოდენობას 20-დან 50-მდე დიაპაზონში.

3.4.18. იპოვის 10-დან 70-მდე იმ რიცხვების ჯამს, რომლებიც არ არიან 8-ის ჯერადი.

continue, break ოპერატორები

შეადგინეთ პროგრამა, რომელიც:

3.5.1. შეკრებს 1-დან 20-მდე რიცხვებს მანამ, სანამ ჯამი არ გახდება 24-ზე მეტი.

3.5.2. დათვლის 4-ის ჯერადი რიცხვებს 6-დან 79-მდე დიაპაზონში მანამ, სანამ 4-ის ჯერადი რიცხვი არ გახდება 50-ზე მეტი.

3.5.3. შეკრებს 9-ის ჯერად რიცხვებს 2-დან 47-მდე დიაპაზონში მანამ, სანამ 9-ის ჯერადი რიცხვი იქნება 30-ზე ნაკლები.

თავი 4. მასივი, სტრიქონი და ბიტობრივი ოპერაცია

ერთგანზომილებიანი მასივი

შეადგინეთ პროგრამა, რომელიც:

4.1.1. იპოვის M_{10} მასივის ელემენტების ჯამს.

4.1.2. იპოვის M_{10} მასივის ელემენტების ნამრავლს.

4.1.3. იპოვის M_{10} მასივის ელემენტების კვადრატების ჯამს.

4.1.4. იპოვის M_{10} მასივის მინიმალურ ელემენტს.

4.1.5. იპოვის M_{10} მასივის მაქსიმალური ელემენტის ინდექსს.

4.1.6. იპოვის M_{10} მასივის ლუწი ელემენტების ჯამს.

4.1.7. იპოვის M_{10} მასივის კენტი ელემენტების რაოდენობას.

4.1.8. იპოვის M_{10} მასივის დადებითი ელემენტების ჯამს.

4.1.9. იპოვის M_{10} მასივის უარყოფითი ელემენტების რაოდენობას.

4.1.10. იპოვის M_{10} მასივის არანულოვანი ელემენტების რაოდენობას.

4.1.11. იპოვის M_{10} მასივის ნულოვანი ელემენტების რაოდენობას.

4.1.12. იპოვის M_{10} მასივის ლუწი ინდექსის მქონე ელემენტების ჯამს.

4.1.13. იპოვის M_{10} მასივის კენტი ინდექსის მქონე ელემენტების რაოდენობას.

4.1.14. იპოვის მაქსიმალურ სხვაობას M_{10} და N_{10} მასივების ელემენტებს შორის.

4.1.15. იპოვის M_{10} მასივის პირველ უარყოფით ელემენტს და მის ინდექსს.

4.1.16. იპოვის M_{10} მასივის პირველი დადებითი ელემენტის ინდექსს.

4.1.17. იპოვის M_{10} მასივის უარყოფითი ელემენტებიდან უდიდესის ინდექსს.

4.1.18. გამოთვლის M_{10} მასივის ელემენტების საშუალო არითმეტიკულს.

4.1.19. გამოთვლის M_{10} მასივის ელემენტების საშუალო გეომეტრიულს.

4.1.20. დათვლის M_{10} მასივის იმ ელემენტების რაოდენობას, რომლებიც მეტია b რიცხვზე.

4.1.21. იპოვის M_{10} მასივის იმ ელემენტების ჯამს, რომლებიც ნაკლებია b რიცხვზე.

4.1.22. განსაზღვრავს თუ რამდენი ელემენტია მოთავსებული M_{10} მასივის ორ მოცემულ ელემენტს შორის.

4.1.23. იპოვის M_{10} მასივის დადებითი ელემენტებიდან უმცირესს.

4.1.24. იპოვის M_{10} მასივის უარყოფითი ელემენტებიდან უდიდესს.

4.1.25. იპოვის იმ ელემენტების ჯამს, რომლებიც მოთავსებულია M_{10} მასივის ორ მოცემულ ელემენტს შორის.

4.1.26. იპოვის იმ ელემენტებს შორის მაქსიმალურს, რომლებიც მოთავსებულია M_{10} მასივის ორ

მოცემულ ელემენტს შორის.
4.1.27. დათვლის M_{10} მასივის იმ ელემენტების რაოდენობას, რომელთა მნიშვნელობები 20-ზე მეტია.
4.1.28. M_{10} მასივიდან N_{10} მასივში გადაწერს იმ ელემენტებს, რომელთა მნიშვნელობები 20-ზე ნაკლებია.
4.1.29. M_{10} მასივში იპოვის იმ ელემენტების რაოდენობას, რომელთა მნიშვნელობები აღემატება საკუთარ ინდექსს.
4.1.30. M_{10} მასივის დადებით ელემენტებს გადაწერს N_{10} მასივში.
4.1.31. M_{10} მასივის 5-ის ჯერად ელემენტებს გადაწერს N_{10} მასივში.
4.1.32. M_{10} მასივის არანულოვან ელემენტებს გადაწერს N_{10} მასივში.
4.1.33. M_{10} მასივის კენტი ინდექსის მქონე ელემენტებს გადაწერს N_5 მასივში.
4.1.34. M_{10} მასივის ლუწი ინდექსის მქონე ელემენტებს გადაწერს N_5 მასივში.
4.1.35. M_{20} მასივის ლუწინდექსიან ელემენტებს გადაწერს A_{10} მასივში, ხოლო კენტინდექსიან ელემენტებს კი - B_{10} მასივში.
4.1.36. M_{10} მასივის დადებითი ელემენტების ინდექსებს გადაწერს N_{10} მასივში.
4.1.37. M_{10} მასივის ლუწ ელემენტებს გადაწერს N_{10} მასივში.
4.1.38. M_{10} მასივის კენტი ინდექსის მქონე ელემენტებს გადაწერს N_5 მასივში.
4.1.39. M_{10} მასივის 3-ის ჯერადი ელემენტების ინდექსებს გადაწერს N_{10} მასივში.
4.1.40. მოახდენს ზრდადობის მიხედვით დალაგებული A_{10} და B_{10} მასივების შერწყმას და შექმნის ახალ ზრდადობის მიხედვით დალაგებულ M_{20} მასივს.
4.1.41. M_{10} მასივში განსაზღვრავს გაორმაგებული კენტი რიცხვების რაოდენობას და ჩაწერს მათ N მასივში.
4.1.42. N მასივში გადაწერს M_{10} მასივის იმ ელემენტებს, რომლებიც 5-ზე გაყოფისას იძლევიან 1-დან 4-ის ტოლ ნაშთს.
4.1.43. M_{10} მასივის უარყოფით ელემენტებს შეცვლის 0-ებით.
4.1.44. M_{10} მასივის კენტ ელემენტებს შეცვლის მათი კვადრატებით.
4.1.45. M_{10} მასივის ელემენტებს უკუ (შებრუნებული) თანმიმდევრობით დაალაგებს.
4.1.46. M_{10} მასივის ელემენტებს დაალაგებს ისე, რომ დასაწყისში განლაგდეს ყველა უარყოფითი რიცხვი რიგითობის შენარჩუნებით, შემდეგ კი დადებითი რიცხვები რიგითობის შენარჩუნებით. დამხმარე მასივი არ გამოიყენოთ.
4.1.47. შექმნის A_{10} მასივს, რომელშიც ჯერ მოთავსებული იქნება M_{10} მასივის არანულოვანი ელემენტები, შემდეგ კი - ნულოვანი.
4.1.48. M_{10} მასივში ელემენტებს დაალაგებს ისე, რომ ჯერ მოთავსდეს არანულოვანი ელემენტები, შემდეგ კი - ნულოვანი.
4.1.49. M_{10} მასივის ელემენტებს ციკლურად დაძრავს მარჯვნივ n ელემენტით.
4.1.50. M_{10} მასივის ელემენტებს ციკლურად დაძრავს მარცხნივ n ელემენტით.
4.1.51. M_{10} მასივში ადგილებს შეუცვლის კენტი და ლუწი ინდექსის მქონე ელემენტებს.
4.1.52. შეამოწმებს ემთხვევა თუ არა M_{10} მასივის პირველი ნახევარი მის მეორე ნახევარს.
4.1.53. შეამოწმებს განსხვავდებიან თუ არა M_{10} და N_{10} მასივები.
4.1.54. განსაზღვრავს შეიცავს თუ არა M_{10} მასივი N_{10} მასივს.
4.1.55. განსაზღვრავს M_{10} მასივის ერთმანეთის მიყოლებით მდებარე ნულის ტოლი ელემენტების ყველაზე გრძელ მიმდევრობას.
4.1.56. დათვლის M_{10} მასივის განსხვავებული ელემენტების რაოდენობას.
4.1.57. დაადგენს არის თუ არა M_{10} მასივში ერთნაირი ელემენტები.
4.1.58. M_{10} მასივში იპოვის ზრდადობით დალაგებულ ყველაზე გრძელ მიმდევრობას.
4.1.59. M_{10} მასივში იპოვის კლებადობით დალაგებულ ყველაზე მოკლე მიმდევრობას.
4.1.60. დათვლის M_{10} მასივის მეზობელ ელემენტებს შორის ნიშანცვლათა რაოდენობას.

4.1.61. იპოვის M_{10} მასივის იმ ელემენტების ინდექსებს, რომელთა შორის სხვაობა უდიდესია.
4.1.62. იპოვის M_{10} მასივის იმ ელემენტების ინდექსებს, რომელთა შორის სხვაობა უმცირესია.
4.1.63. M_{10} მასივში იპოვის იმ 5-ელემენტიან მიმდევრობას, რომლის ელემენტების ჯამი მინიმალურია.
ბ. კენტი ინდექსის მქონე რიცხვებს შორის უდიდესს;
გ. ლუწი ინდექსის მქონე რიცხვებს შორის უმცირესს.

ორგანზომილებიანი მასივი

შეადგინეთ პროგრამა, რომელიც:
4.2.1. იპოვის $A_{5,5}$ მასივის მაქსიმალურ ელემენტს.
4.2.2. იპოვის $A_{5,5}$ მასივის მინიმალური ელემენტის ინდექსებს.
4.2.3. იპოვის $A_{5,5}$ მასივის ელემენტების ჯამს.
4.2.4. იპოვის $A_{5,5}$ მასივის ელემენტების ნამრავლს.
4.2.5. იპოვის $A_{5,5}$ მასივის დადებითი ელემენტების ჯამს.
4.2.6. იპოვის $A_{5,5}$ მასივის უარყოფითი ელემენტების რაოდენობას.
4.2.7. იპოვის $A_{5,5}$ მასივის კენტი ელემენტების ჯამს.
4.2.8. იპოვის $A_{5,5}$ მასივის ლუწი ელემენტების რაოდენობას.
4.2.9. იპოვის $A_{5,5}$ მასივის არანულოვანი ელემენტების ნამრავლს.
4.2.10. იპოვის $A_{5,5}$ მასივის ნულოვანი ელემენტების რაოდენობას.
4.2.11. იპოვის $A_{5,5}$ მასივის მთავარი დიაგონალის ელემენტების ჯამს.
4.2.12. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის ელემენტების ნამრავლს.
4.2.13. იპოვის $A_{5,5}$ მასივის მთავარი დიაგონალის მინიმალურ ელემენტს.
4.2.14. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის მაქსიმალური ელემენტის ინდექსებს.
4.2.15. იპოვის $A_{5,5}$ მასივის მთავარი დიაგონალის 6-ის ჯერადი ელემენტების რაოდენობას.
4.2.16. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის დადებითი ელემენტების ჯამს.
4.2.17. იპოვის $A_{5,5}$ მასივის 4-ის ჯერადი ელემენტების რაოდენობას.
4.2.18. დათვლის $A_{5,5}$ მასივის იმ ელემენტების რაოდენობას, რომელთა მნიშვნელობები 20-ზე მეტია.
4.2.19. $A_{5,5}$ მასივიდან M_{25} მასივში გადაწერს იმ ელემენტებს, რომელთა მნიშვნელობები 20-ზე ნაკლებია.
4.2.20. $A_{5,5}$ მასივის ლუწ ელემენტებს გადაწერს B_{25} მასივში.
4.2.21. $A_{5,5}$ მასივის არანულოვან ელემენტებს გადაწერს B_{25} მასივში.
4.2.22. $A_{5,5}$ მასივის თითოეული სვეტის ელემენტების ჯამს ჩაწერს B_5 მასივში.
4.2.23. $A_{5,5}$ მასივის თითოეული სტრიქონის ელემენტების ნამრავლს ჩაწერს B_5 მასივში.
4.2.24. დათვლის $A_{5,5}$ მასივის თითოეული სვეტის დადებითი ელემენტების რაოდენობას და ჩაწერს მათ B_5 მასივში.
4.2.25. იპოვის $A_{5,5}$ მასივის თითოეული სტრიქონის კენტი ელემენტების ჯამს და ჩაწერს მათ B_5 მასივში.
4.2.26. იპოვის $A_{5,5}$ მასივის თითოეული სვეტის მაქსიმალურ ელემენტს და ჩაწერს მათ B_5 მასივში.
4.2.27. იპოვის $A_{5,5}$ მასივის თითოეული სტრიქონის მინიმალურ ელემენტს და ჩაწერს მათ B_5 მასივში.
4.2.28. იპოვის $A_{5,5}$ მასივის მთავარი დიაგონალის მაქსიმალურ ელემენტს და ამ ელემენტის შემცველ სტრიქონს გადაწერს B_5 მასივში.
4.2.29. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის მინიმალურ ელემენტს და ამ ელემენტის შემცველ სვეტს გადაწერს B_5 მასივში.
4.2.30. იპოვის $A_{5,5}$ მასივის იმ სტრიქონს, რომელიც შეიცავს ლუწი ელემენტების მაქსიმალურ

რაოდენობას და ამ სტრიქონს გადაწერს $B_{5,5}$ მასივში.

4.2.31. იპოვის $A_{5,5}$ მასივის იმ სვეტს, რომელიც შეიცავს კენტი ელემენტების მინიმალურ რაოდენობას და ამ სვეტს გადაწერს B_5 მასივში.

4.2.32. იპოვის $A_{5,5}$ მასივის მაქსიმალური ელემენტის შემცველ სტრიქონს და გადაწერს მას B_5 მასივში.

4.2.33. იპოვის $A_{5,5}$ მასივის მინიმალური ელემენტის შემცველ სვეტს და გადაწერს მას B_5 მასივში.

4.2.34. $A_{5,5}$ მასივის მთავარი დიაგონალის ელემენტებს B_5 მასივში გადაწერს.

4.2.35. $A_{5,5}$ მასივის არამთავარი დიაგონალის ელემენტებს B_5 მასივში გადაწერს.

4.2.36. იანგარიშებს $A_{5,5}$ მასივის ლუწი ნომრების მქონე სვეტების ელემენტების საშუალო არითმეტიკულს და ჩაწერს მათ B_3 მასივში.

4.2.37. იანგარიშებს $A_{5,5}$ მასივის კენტი ნომრების მქონე სტრიქონების ელემენტების საშუალო გეომეტრიულს და ჩაწერს მათ B_2 მასივში.

4.2.38. იპოვის $A_{5,5}$ მასივის იმ სტრიქონების ინდექსებს, რომელთა ყველა ელემენტი ნულის ტოლია. ინდექსები ჩაწერეთ B_5 მასივში.

4.2.39. იპოვის $A_{5,5}$ მასივის იმ სვეტების ინდექსებს, რომელთა ყველა ელემენტი კენტია. ინდექსები ჩაწერეთ B_5 მასივში.

4.2.40. იპოვის $A_{5,5}$ მასივის იმ სტრიქონების ინდექსებს, რომელთა ელემენტები ქმნიან მონოტონურად ზრდად მიმდევრობას. ინდექსები ჩაწერეთ B_5 მასივში.

4.2.41. იპოვის $A_{5,5}$ მასივის იმ სვეტების ინდექსებს, რომელთა ელემენტები ქმნიან მონოტონურად კლებად მიმდევრობას. ინდექსები ჩაწერეთ B_5 მასივში.

4.2.42. $A_{5,5}$ მასივიდან წაშლის იმ სვეტსა და სტრიქონს, რომელთა გადაკვეთაზე მდებარეობს მატრიცის მინიმალური ელემენტი.

4.2.43. $A_{5,5}$ მასივიდან წაშლის ნულების შემცველ სტრიქონს.

4.2.44. $A_{5,5}$ მასივიდან წაშლის ნულების შემცველ სვეტს.

4.2.45. $A_{5,5}$ მასივის ნულოვან ელემენტებს შეცვლის 1-ებით.

4.2.46. $A_{5,5}$ მასივში ადგილებს შეუცვლის მითითებული ნომრის მქონე სვეტსა და უკანასკნელ სვეტს.

4.2.47. $A_{5,5}$ მასივში ადგილებს შეუცვლის მითითებული ნომრის მქონე სტრიქონსა და პირველ სტრიქონს.

4.2.48. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ძვრას ზევით ერთი სტრიქონით. ეს იმას ნიშნავს, რომ მეორე სტრიქონი დაიკავებს პირველის ადგილს, მესამე სტრიქონი დაიკავებს მეორის ადგილს და ა.შ. უკანასკნელი სტრიქონი შეივსება ნულებით.

4.2.49. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ძვრას ზევით ორი სტრიქონით.

4.2.50. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ძვრას ქვევით ერთი სტრიქონით. ეს იმას ნიშნავს, რომ პირველი სტრიქონი დაიკავებს მეორის ადგილს, მეორე სტრიქონი დაიკავებს მესამის ადგილს და ა.შ. პირველი სტრიქონი შეივსება ნულებით.

4.2.51. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ძვრას ქვევით ორი სტრიქონით.

4.2.52. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ციკლისებურ ძვრას ზევით ერთი სტრიქონით. ეს იმას ნიშნავს, რომ მეორე სტრიქონი დაიკავებს პირველის ადგილს, მესამე სტრიქონი დაიკავებს მეორის ადგილს და ა.შ. პირველი სტრიქონი დაიკავებს უკანასკნელის ადგილს.

4.2.53. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ციკლისებურ ძვრას ზევით ორი სტრიქონით.

4.2.54. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ციკლისებურ ძვრას ქვევით ერთი სტრიქონით. ეს იმას ნიშნავს, რომ პირველი სტრიქონი დაიკავებს მეორის ადგილს, მეორე სტრიქონი დაიკავებს მესამის ადგილს და ა.შ. უკანასკნელი სტრიქონი დაიკავებს პირველის ადგილს.

4.2.55. შეასრულებს $A_{5,5}$ მასივის სტრიქონების ციკლისებურ ძვრას ქვევით ორი სტრიქონით.

4.2.56. შეასრულებს $A_{5,5}$ მასივის სვეტების ძვრას მარჯვნივ ერთი სვეტით.

4.2.57. შეასრულებს $A_{5,5}$ მასივის სვეტების ძვრას მარჯვნივ ორი სვეტით.

4.2.58. შეასრულებს $A_{5,5}$ მასივის სვეტების ძვრას მარცხნივ ერთი სვეტით.
4.2.59. შეასრულებს $A_{5,5}$ მასივის სვეტების ძვრას მარცხნივ ორი სვეტით.
4.2.60. შეასრულებს $A_{5,5}$ მასივის სვეტების ციკლისებურ ძვრას მარცხნივ ერთი სვეტით.
4.2.61. შეასრულებს $A_{5,5}$ მასივის სვეტების ციკლისებურ ძვრას მარცხნივ ორი სვეტით.
4.2.62. შეასრულებს $A_{5,5}$ მასივის სვეტების ციკლისებურ ძვრას მარჯვნივ ერთი სვეტით.
4.2.63. შეასრულებს $A_{5,5}$ მასივის სვეტების ციკლისებურ ძვრას მარჯვნივ ორი სვეტით.
4.2.64. იპოვის $A_{5,5}$ მასივის იმ სტრიქონის ინდექსს, რომელიც შეიცავს 8-ის ჯერადი ელემენტების მაქსიმალურ რაოდენობას.
4.2.65. იპოვის $A_{5,5}$ მასივის იმ სვეტის ინდექსს, რომელიც შეიცავს დადებითი ელემენტების მინიმალურ რაოდენობას.
4.2.66. იპოვის $A_{5,5}$ მასივის იმ ელემენტების ჯამს, რომლებიც მთავარი დიაგონალის ზემოთაა მოთავსებული.
4.2.67. იპოვის $A_{5,5}$ მასივის იმ ელემენტების ნამრავლს, რომლებიც მთავარი დიაგონალის ქვემოთაა მოთავსებული.
4.2.68. იპოვის $A_{5,5}$ მასივის ორი ტოლი ელემენტის ინდექსებს.
4.2.69. იპოვის $A_{5,5}$ მასივის იმ ელემენტს, რომელიც ყველა სტრიქონში შედის.
4.2.70. იპოვის $A_{5,5}$ მასივის იმ ელემენტს, რომელიც ყველა სვეტში შედის.
4.2.71. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის ზემოთ მოთავსებული ელემენტების ჯამს.
4.2.72. იპოვის $A_{5,5}$ მასივის არამთავარი დიაგონალის ქვემოთ მოთავსებული ელემენტების ჯამს.
4.2.73. იპოვის $A_{5,5}$ მასივის იმ სტრიქონის ნომერს, რომლის ელემენტების ჯამი აღემატება დანარჩენი სტრიქონების ელემენტების ჯამს.
4.2.74. იპოვის $A_{5,5}$ მასივის იმ სვეტის ნომერს, რომლის ელემენტების ჯამი აღემატება დანარჩენი სვეტების ელემენტების ჯამს.
4.2.75. განსაზღვრავს არის თუ არა $A_{5,5}$ მასივში ერთნაირი სვეტები.
4.2.76. განსაზღვრავს არის თუ არა $A_{5,5}$ მასივში ერთნაირი სტრიქონები.
4.2.77. $A_{5,5}$ მასივიდან წაშლის იმ სვეტსა და სტრიქონს, რომელთა გადაკვეთაზე მდებარეობს მთავარი დიაგონალის მაქსიმალური ელემენტი.
4.2.78. $A_{5,5}$ მასივის რომელიმე სვეტსა და სტრიქონს შეავსებს ნულებით მათ გადაკვეთაზე მდებარე ელემენტის გარდა.
4.2.79. იპოვის $A_{5,5}$ მასივის უნაგირა წერტილს. უნაგირა წერტილი არის მასივის ის ელემენტი, რომელიც უმცირესია თავის სტრიქონში და უდიდესია თავის სვეტში.
4.2.80. იპოვის $A_{5,5}$ მასივის იმ ელემენტების ჯამს, რომლებიც არამთავარი დიაგონალის ქვემოთაა მოთავსებული.
4.2.81. იპოვის $A_{5,5}$ მასივის იმ ელემენტების ჯამს, რომლებიც არამთავარი დიაგონალის ზემოთაა მოთავსებული.

თავი 6. კლასი, ინკაფსულაცია, ფუნქცია

კლასი. ინკაფსულაცია

6.1.1. შექმენით თვითმფრინავის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: ავზის ტევადობა და მანძილი, რომელსაც თვითმფრინავი 1 ლიტრი საწვავით გაიფრენს; საერთო წვდომის ცვლადებს: მგზავრების რაოდენობა და გაყიდული ბილეთების რაოდენობა.
6.1.2. შექმენით სტუდენტის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: გვარი, სახელი და ასაკი; საერთო წვდომის ცვლადებს: უნივერსიტეტის დასახელება და კურსი.
6.1.3. შექმენით მატარებლის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: ვაგონების რაოდენობასა და მგზავრების რაოდენობას 1 ვაგონში; საერთო წვდომის ცვლადებს: ბილეთების

ფასს და გაყიდული ბილეთების რაოდენობას.

6.1.4. შექმენით სამკუთხედის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: პერიმეტრი და ფართობი; საერთო წვდომის ცვლადებს: სამკუთხედის სამივე გვერდის ზომებს.

ფუნქცია

6.2.1. შექმენით სტუდენტის კლასი, რომელიც შეიცავს ღია ფუნქციას. ფუნქციას გადაეცემა სტუდენტის ნიშნები, რომელიც 10-ელემენტიანი მთელრიცხვა მასივია, ფუნქცია გამოთვლის და აბრუნებს ნიშნების საშუალო არითმეტიკულს.

6.2.2. შექმენით სტუდენტის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: გვარი, სახელი და ასაკი; პრივატულ ფუნქციას, რომელიც პრივატულ ცვლადებს მნიშვნელობებს ანიჭებს; ღია ფუნქციებს: პირველი ფუნქცია იძახებს პრივატულ ფუნქციას და გადასცემს მნიშვნელობებს პრივატული ცვლადებისათვის მისანიჭებლად; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები.

6.2.3. შექმენით მატარებლის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: ვაგონების რაოდენობასა და მგზავრების რაოდენობას 1 ვაგონში; საერთო წვდომის ცვლადებს: ბილეთების ფასს და გაყიდული ბილეთების რაოდენობას. შეიცავს ღია ფუნქციებს: პირველი ფუნქცია პრივატულ და ღია ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული და ღია ცვლადების მნიშვნელობები; მესამე ფუნქცია გამოთვლის და აბრუნებს ბილეთების გაყიდვით მიღებულ თანხას.

6.2.4. შექმენით თვითმფრინავის კლასი, რომელიც შეიცავს ღია ცვლადებს: ავზის ტევადობა და მანძილი, რომელსაც გაიფრენს თვითმფრინავი 1 ლიტრი საწვავით; ღია ფუნქციებს: პირველი ფუნქცია ღია ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს ღია ცვლადების მნიშვნელობები; მესამე ფუნქცია გამოთვლის და აბრუნებს თვითმფრინავის მიერ სავსე ავზით გავლილ მანძილს.

6.2.5. შექმენით მართკუთხედის კლასი, რომელიც შეიცავს ღია ცვლადებს: მართკუთხედის ოთხივე გვერდის ზომებს; პრივატულ ფუნქციას, რომელიც ღია ცვლადებს მნიშვნელობებს ანიჭებს; ღია ფუნქციებს: პირველი ფუნქცია იძახებს პრივატულ ფუნქციას, რომელიც პრივატულ ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები; მესამე ფუნქცია გამოთვლის და აბრუნებს მართკუთხედის ფართობს.

კონსტრუქტორი

6.3.1. შექმენით თვითმფრინავის კლასი, რომელიც შეიცავს პრივატულ ცვლადებს: ავზის ტევადობა და მანძილი, რომელსაც გაიფრენს თვითმფრინავი 1 ლიტრი საწვავით; ღია ფუნქციებს: პირველი ფუნქციაა კონსტრუქტორი, რომელიც პრივატულ ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები; მესამე ფუნქცია გამოთვლის და აბრუნებს თვითმფრინავის მიერ სავსე ავზით გავლილ მანძილს.

6.3.2. შექმენით სამკუთხედის კლასი, რომელიც შეიცავს ღია ცვლადებს: სამკუთხედის სამივე გვერდის ზომებს; პრივატულ ცვლადებს: სამკუთხედის პერიმეტრსა და ფართობს; ღია ფუნქციებს: პირველი ფუნქციაა კონსტრუქტორი, რომელიც ღია და პრივატულ ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები.

6.3.3. შექმენით მართკუთხედის კლასი, რომელიც შეიცავს ღია ცვლადებს: მართკუთხედის ოთხივე გვერდის ზომებს; პრივატულ ცვლადებს: მართკუთხედის ფართობს და პერიმეტრს; ღია ფუნქციებს: პირველი ფუნქციაა კონსტრუქტორი, რომელიც ღია და პრივატულ ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები.

this მიმთითებელი

6.4.1. შექმენით კლასი, რომლის ცვლადებსა და ფუნქციის პარამეტრებს ერთნაირი სახელები აქვთ. პარამეტრების მნიშვნელობები მიანიჭეთ კლასის ამავე სახელის მქონე ცვლადებს. ფუნქცია აბრუნებს კლასის ცვლადების ჯამს.

6.4.2. შექმენით კლასი, რომლის ცვლადებსა და ფუნქციის პარამეტრებს ერთნაირი სახელები აქვთ. პარამეტრების მნიშვნელობები მიანიჭეთ კლასის ამავე სახელის მქონე ცვლადებს. ფუნქცია აბრუნებს კლასის ცვლადების ნამრავლს.

6.4.3. შექმენით კლასი, რომელშიც განსაზღვრულ მასივსა და ფუნქციის პარამეტრს ერთნაირი სახელები აქვთ. პარამეტრის მნიშვნელობა მიანიჭეთ კლასის ამავე სახელის მქონე მასივს. ფუნქცია აბრუნებს მასივის პირველ უარყოფით ელემენტს.

6.4.4. შექმენით კლასი, რომელშიც განსაზღვრულ მასივსა და ფუნქციის პარამეტრს ერთნაირი სახელები აქვთ. პარამეტრის მნიშვნელობა მიანიჭეთ კლასის ამავე სახელის მქონე მასივს. ფუნქცია აბრუნებს მასივის ელემენტებს შორის მინიმალურს.

static მოდიფიკატორი

6.5.1. შექმენით Otxkutxedi კლასი, რომელიც შეიცავს: სტატიკურ ცვლადებს - ოთხკუთხედის გვერდებს, რომელთა ინიციალიზებას კონსტრუქტორი ასრულებს; ჩვეულებრივ ფუნქციას, რომელიც ოთხკუთხედის პერიმეტრს ანგარიშობს; სტატიკურ ფუნქციას, რომელიც ჩვეულებრივ ფუნქციას იძახებს.

6.5.2. შექმენით ChemiKlasi კლასი, რომელიც შეიცავს: ერთგანზომილებიან სტატიკურ მასივს, რომლის ინიციალიზებას კონსტრუქტორი ახდენს; ჩვეულებრივ ფუნქციას, რომელიც ერთგანზომილებიანი სტატიკური მასივის კენტი ელემენტების ჯამს გასცემს; სტატიკურ ფუნქციას, რომელიც ჩვეულებრივ ფუნქციას იძახებს და გასცემს შედეგს.

6.5.3. შექმენით ChemiKlasi კლასი, რომელიც შეიცავს: ორგანზომილებიან სტატიკურ მასივს, რომლის ინიციალიზებას კონსტრუქტორი ახდენს; ჩვეულებრივ ფუნქციას, რომელიც ორგანზომილებიანი სტატიკური მასივის კენტი ელემენტების რაოდენობას გასცემს; სტატიკურ ფუნქციას, რომელიც ჩვეულებრივ ფუნქციას იძახებს და გასცემს შედეგს.

შემთხვევითი რიცხვების გენერატორი

შეადგინეთ პროგრამა, რომელიც:

6.6.1. მოახდენს [1,100] ინტერვალში მოთავსებული 20 შემთხვევითი რიცხვის გენერირებას.

6.6.2. დათვლის თუ რამდენჯერ გვხვდება q რიცხვი [1,1000] ინტერვალში 50 მცდელობის შემდეგ.

6.6.3. ჯერ მოახდენს 100 მთელი რიცხვის გენერირებას [1,100] ინტერვალში, შემდეგ კი დათვლის თუ რამდენჯერ გამოჩნდა თითოეული რიცხვი ამ ინტერვალში.

6.6.4. განსაზღვრავს თუ რამდენი მთელი რიცხვის გენერირება უნდა მოვახდინოთ [1,30] ინტერვალიდან, რომ მოცემული R რიცხვი მათ შორის 7-ჯერ შეგვხვდეს.

6.6.5. განსაზღვრავს თუ რამდენი ასანთი უნდა ამოვიღოთ ასანთის 5 ყუთიდან იმისათვის, რომ დაცარიელდეს ერთ-ერთი. თითო ყუთში 20 ასანთია.

თავი 7. პოლიმორფიზმი

ფუნქციისათვის ობიექტის გადაცემა. ობიექტის დაბრუნება

7.1.1. შექმენით Samkutxedi კლასი. მასში შექმენით ფუნქცია, რომელსაც გადაეცემა ამავე ტიპის ობიექტი და რომელიც შეასრულებს ამ ობიექტის ფართობის გამოთვლას.

7.1.2. შექმენით Otxkutxedi კლასი. მასში შექმენით ფუნქცია, რომელსაც გადაეცემა ამავე ტიპის

ობიექტი და რომელიც შეასრულებს ამ ობიექტის პერიმეტრის გამოთვლას.

7.1.3. შექმენით Kvadrati_1 კლასი. მასში გამოაცხადეთ ფართობი და პერიმეტრი, და კონსტრუქტორი რომელიც ამ ცვლადებს მნიშვნელობებს ანიჭებს. შექმენით Kvadrati_2 კლასი. მასში გამოაცხადეთ ფუნქცია, რომელიც ანგარიშობს სამკუთხედის ფართობს და პერიმეტრს, და აბრუნებს ამ პარამეტრების მქონე Kvadrati_1 ტიპის ობიექტს.

7.1.4. შექმენით ChemiKlasi_1 კლასი. მასში გამოაცხადეთ ორი ცვლადი მასივის მინიმალური და მაქსიმალური ელემენტების მოსათავსებლად და კონსტრუქტორი, რომელიც ამ მასივისა ცვლადის ინიციალიზებას ასრულებს. შექმენით ChemiKlasi_2 კლასი. მასში გამოაცხადეთ ფუნქცია, რომლის პარამეტრია ერთგანზომილებიანი მასივი და რომელიც პოულობს ამ მასივის ელემენტებს შორის მინიმალურს და მაქსიმალურს და აბრუნებს ამ პარამეტრების მქონე ChemiKlasi_1 ტიპის ობიექტს.

ფუნქციების გადატვირთვა. პოლიმორფიზმი

7.2.1. შეადგინეთ სამკუთხედის კლასი, რომელშიც განსაზღვრულია ერთი და იმავე სახელის მქონე 2 ფუნქცია. პირველ ფუნქციას 2 მთელრიცხვა პარამეტრი აქვს: სამკუთხედის სიმაღლე და ფუძე და აბრუნებს მართკუთხა სამკუთხედის ფართობს. მეორე ფუნქციას 3 მთელრიცხვა პარამეტრი აქვს: სამკუთხედის გვერდები და აბრუნებს სამკუთხედის პერიმეტრს.

7.2.2. შეადგინეთ ფიგურის კლასი, რომელშიც განსაზღვრულია ერთი და იმავე სახელის მქონე 3 ფუნქცია. პირველ ფუნქციას ერთი მთელრიცხვა პარამეტრი აქვს და აბრუნებს კვადრატის პერიმეტრს. მეორე ფუნქციას ორი მთელრიცხვა პარამეტრი აქვს და აბრუნებს ოთხკუთხედის პერიმეტრს. მესამე ფუნქციას სამი მთელრიცხვა პარამეტრი აქვს და აბრუნებს სამკუთხედის პერიმეტრს.

7.2.3. შექმენით ავტომობილის კლასი, რომელშიც განსაზღვრულია ერთი და იმავე სახელის მქონე 2 ფუნქცია. პირველ ფუნქციას 2 მთელრიცხვა პარამეტრი აქვს: ავზის ტევადობა და მანძილი, რომელსაც გაივლის ავტომობილი 1 ლიტრი საწვავით, და აბრუნებს ავტომობილის მიერ სავსე ავზით გავლილ მანძილს. მეორე ფუნქციას 2 წილადი პარამეტრი აქვს: მაქსიმალური სიჩქარე და მოძრაობის დრო, და აბრუნებს მანძილს, რომელსაც გაივლის ავტომობილი მითითებული დროის განმავლობაში მაქსიმალური სიჩქარით მოძრაობისას.

7.2.4. შეადგინეთ მატარებლის კლასი, რომელშიც განსაზღვრულია ერთი და იმავე სახელის მქონე 2 ფუნქცია. პირველ ფუნქციას 2 მთელრიცხვა პარამეტრი აქვს: ვაგონების რაოდენობა და ერთ ვაგონში მგზავრების რაოდენობა, და აბრუნებს მგზავრების საერთო რაოდენობას. მეორე ფუნქციას 2 წილადი პარამეტრი აქვს: 1 კილომეტრის გავლისას დახარჯული ელექტროენერგია და გავლილი მანძილი, და აბრუნებს მატარებლის მიერ მითითებული მანძილის გავლისას დახარჯულ ელექტროენერგიას.

კონსტრუქტორის გადატვირთვა

7.3.1. შექმენით კლასი, რომელიც მთელი ტიპის Min ცვლადს შეიცავს. კლასის I კონსტრუქტორი, რომელსაც გადაეცემა ერთგანზომილებიანი მთელრიცხვა მასივი, Min ცვლადს ანიჭებს მასივის მინიმალურ ელემენტს. კლასის II კონსტრუქტორს პარამეტრად გადაეცემა ამავე კლასის ობიექტი, რომლის საშუალებით მოხდება ამავე კლასის Min ცვლადის ინიციალიზება.

7.3.2. შექმენით კლასი, რომელიც სამ გვერდს, პერიმეტრს, ფართობსა და სამ კონსტრუქტორს შეიცავს. I კონსტრუქტორს 1 მთელი ტიპის პარამეტრი აქვს და ქმნის კვადრატს (გამოთვლის პერიმეტრსა და ფართობს). II კონსტრუქტორს 2 მთელი ტიპის პარამეტრი აქვს და ქმნის მართკუთხედს. III კონსტრუქტორს 3 მთელი ტიპის პარამეტრი აქვს და ქმნის სამკუთხედს.

თავი 8. მემკვიდრეობითობა

კლასები. მემკვიდრეობითობა. protected მოდიფიკატორი

8.1.1. შექმენით სამკუთხედის საბაზო კლასი, რომელიც შეიცავს სამ პრივატულ ცვლადს - სამკუთხედის გვერდებს. შექმენით სამკუთხედის მემკვიდრე კლასი, რომელიც დამატებით შეიცავს პრივატულ ცვლადს - სამკუთხედის პერიმეტრს და ორ ღია ფუნქციას. პირველი ფუნქცია გამოთვლის და აბრუნებს სამკუთხედის ფართობს. მეორე ფუნქცია გამოთვლის სამკუთხედის პერიმეტრს.

8.1.2. შექმენით მართკუთხედის საბაზო კლასი, რომელიც შეიცავს დაცულ ცვლადს: მართკუთხედის ფუძეს. შექმენით მართკუთხედის მემკვიდრე კლასი, რომელიც დამატებით შეიცავს: პრივატულ ცვლადს - მართკუთხედის სიმაღლეს; ღია კონსტრუქტორს, რომელიც მართკუთხედის გვერდებს მნიშვნელობებს ანიჭებს; ღია ფუნქციას, რომელიც გამოთვლის და აბრუნებს მართკუთხედის ფართობს.

8.1.3. შექმენით მართკუთხედის საბაზო კლასი, რომელიც შეიცავს დაცულ ცვლადს: მართკუთხედის ფუძეს; ღია ფუნქციებს: პირველი ფუნქცია არის კონსტრუქტორი, რომელიც დაცულ ცვლადს მნიშვნელობას ანიჭებს; მეორე ფუნქციას გამოაქვს დაცული ცვლადის მნიშვნელობა. შექმენით მართკუთხედის მემკვიდრე კლასი, რომელიც დამატებით შეიცავს: პრივატულ ცვლადს - მართკუთხედის სიმაღლეს; ღია ფუნქციას, რომელიც გამოთვლის და აბრუნებს მართკუთხედის ფართობს.

8.1.4. შექმენით პიროვნების საბაზო კლასი, რომელიც შეიცავს დაცულ ცვლადებს: გვარს, სახელსა და ასაკს; ღია ფუნქციებს: პირველი ფუნქცია არის კონსტრუქტორი, რომელიც დაცულ ცვლადებს მნიშვნელობებს ანიჭებს; მეორე ფუნქციას გამოაქვს პრივატული ცვლადების მნიშვნელობები. შექმენით ექიმის მემკვიდრე კლასი, რომელიც დამატებით შეიცავს: ღია ცვლადებს: ექიმის ასაკს, განყოფილების დასახელებას, თანამდებობას, საავადმყოფოს დასახელებას, სტაჟს.

ვირტუალური ფუნქციები

8.2.1. შექმენით გეომეტრიული ფიგურის საბაზო კლასი. ის შეიცავს Perimetri ფუნქციას, რომელიც გეომეტრიული ფიგურის პერიმეტრს ანგარიშობს. შექმენით საბაზო კლასის მემკვიდრე კლასი. მასში შექმენით Perimetri ფუნქციის გადატვირთული ვერსია, რომელიც ოთხკუთხედის პერიმეტრს გამოთვლის.

8.2.2. შექმენით საბაზო კლასი. ის შეიცავს Funqcia1 ფუნქციას, რომელიც გასცემს ერთგანზომილებიანი მასივის ელემენტების ჯამს. შექმენით მემკვიდრე კლასი. მასში შექმენით Funqcia1 ფუნქციის გადატვირთული ვერსია, რომელიც გამოთვლის და გასცემს ერთგანზომილებიანი მასივის ელემენტების ნამრავლს.

8.2.3. შექმენით საბაზო კლასი. ის შეიცავს Funqcia1 ფუნქციას, რომელიც გასცემს სტრიქონში სასვენი ნიშნების რაოდენობას. შექმენით მემკვიდრე კლასი. მასში შექმენით Funqcia1 ფუნქციის გადატვირთული ვერსია, რომელიც გამოთვლის და გასცემს სტრიქონში ხმოვნების რაოდენობას.

8.2.4. შექმენით საბაზო კლასი. ის შეიცავს Funqcia1 ფუნქციას, რომელიც გასცემს ორგანზომილებიანი მასივის უარყოფით ელემენტებს შორის მინიმალურს. შექმენით მემკვიდრე კლასი. მასში შექმენით Funqcia1 ფუნქციის გადატვირთული ვერსია, რომელიც გამოთვლის და გასცემს ორგანზომილებიანი მასივის უარყოფით ელემენტებს შორის მაქსიმალურს.

აბსტრაქტული კლასები და ფუნქციები

8.3.1. შექმენით გეომეტრიული ფიგურის აბსტრაქტული საბაზო კლასი, რომელიც შეიცავს Perimetri აბსტრაქტულ ფუნქციას. შექმენით საბაზო კლასის პირველი მემკვიდრე კლასი. მასში

მოხდინეთ Perimetri აბსტრაქტული ფუნქციის რეალიზება, რომელიც ოთხკუთხედის პერიმეტრს გამოთვლის. შექმენით საბაზო კლასის მეორე მემკვიდრე კლასი. მასში მოხდინეთ Perimetri აბსტრაქტული ფუნქციის რეალიზება, რომელიც სამკუთხედის პერიმეტრს გამოთვლის.

8.3.2. შექმენით მატარებლის აბსტრაქტული საბაზო კლასი, რომელიც შეიცავს Gamotvla აბსტრაქტულ ფუნქციას. შექმენით საბაზო კლასის პირველი მემკვიდრე კლასი. მასში მოხდინეთ Gamotvla აბსტრაქტული ფუნქციის რეალიზება, რომელიც გამოთვლის მატარებლის მიერ გავლილ მანძილს (ვიცით 1 სთ-ში რამდენ კილომეტრს გადის და რამდენ საათს მოძრაობდა). შექმენით საბაზო კლასის მეორე მემკვიდრე კლასი. მასში მოხდინეთ Gamotvla აბსტრაქტული ფუნქციის რეალიზება, რომელიც გამოთვლის მატარებლის მიერ დახარჯულ ელექტროენერგიას (ვიცით 1 კმ-ის გავლისას რამდენ ენერგიას ხარჯავს და რამდენი კილომეტრი გაიარა).

8.3.3. შექმენით ტელევიზორის აბსტრაქტული საბაზო კლასი, რომელიც შეიცავს Funqcia აბსტრაქტულ ფუნქციას. შექმენით საბაზო კლასის მემკვიდრე კლასი. მასში მოხდინეთ Funqcia აბსტრაქტული ფუნქციის რეალიზება, რომელიც გამოთვლის ტელევიზორის მიერ დახარჯულ ელექტროენერგიას (ვიცით 1 სთ-ის მუშაობისას რამდენ ენერგიას ხარჯავს და რამდენი საათი იმუშავა).

8.3.4. შექმენით თანამშრომლის აბსტრაქტული საბაზო კლასი, რომელიც შეიცავს Funqcia აბსტრაქტულ ფუნქციას. შექმენით საბაზო კლასის მემკვიდრე კლასი. მასში მოხდინეთ Funqcia აბსტრაქტულ ფუნქციის რეალიზება, რომელიც გამოთვლის თანამშრომლის მიერ 1 წლის განმავლობაში აღებული ხელფასის ჯამს (ვიცით თანამშრომლის ყოველთვიური ხელფასი).

თავი 9. თვისება, ინტერფეისი, დელეგატი, მოვლენა და სახელების სივრცე

თვისება

9.1.1. შექმენით კლასი, რომელშიც პრივატული მთელი ტიპის ცვლადია გამოცხადებული. მასთან მიმართვისათვის გამოიყენეთ თვისება, რომელშიც set და get ფუნქციებია განსაზღვრული. თვისება პრივატულ ცვლადს დადებით მნიშვნელობებს ანიჭებს.

9.1.2. შექმენით კლასი, რომელშიც პრივატული მთელი ტიპის ცვლადია გამოცხადებული. მასთან მიმართვისათვის გამოიყენეთ თვისება, რომელშიც set და get ფუნქციებია განსაზღვრული. თვისება პრივატულ ცვლადს კენტ მნიშვნელობებს ანიჭებს.

9.1.3. შექმენით კლასი, რომელშიც პრივატული მთელი ტიპის ცვლადია გამოცხადებული. მასთან მიმართვისათვის გამოიყენეთ თვისება, რომელშიც set და get ფუნქციებია განსაზღვრული. თვისება პრივატულ ცვლადს 5-ის ჯერად მნიშვნელობებს ანიჭებს.

სკალარული, ინდექსირებული და სტატიკური თვისება

9.2.1. შექმენით კლასი, რომელშიც გამოცხადებულია 5 მთელი ტიპის ცვლადი. მათ მიანიჭეთ მნიშვნელობები სახელდებული ინდექსირებადი თვისების გამოყენებით.

9.2.2. შექმენით კლასი, რომელშიც გამოცხადებულია წილადი ტიპის ცვლადი. მას მიანიჭეთ უარყოფითი მნიშვნელობები სტატიკური თვისების გამოყენებით.

9.2.3. შექმენით კლასი, რომელშიც გამოცხადებულია 5 ლოგიკური ცვლადი. მათ მიანიჭეთ მნიშვნელობები სახელდებული ინდექსირებადი თვისების გამოყენებით.

9.2.4. შექმენით კლასი, რომელშიც გამოცხადებულია სტრიქონი. მას მიანიჭეთ მნიშვნელობები სტატიკური თვისების გამოყენებით.

9.2.5. შექმენით კლასი, რომელშიც გამოცხადებულია მთელრიცხვა ერთგანზომილებიანი

მასივი. სახელდებული ინდექსირებადი თვისების გამოყენებით იპოვეთ მასივის ელემენტების ჯამი.

9.2.6. შექმენით კლასი, რომელშიც გამოცხადებულია წილადების ორგანზომილებიანი მასივი. სახელდებული ინდექსირებადი თვისების გამოყენებით იპოვეთ მასივის მინიმალური ელემენტი.

9.2.7. შექმენით კლასი, რომელშიც გამოცხადებულია 5 მთელი ტიპის ცვლადი. ცვლადებს მიანიჭეთ ლუწი მნიშვნელობები ნაგულისხმევი ინდექსირებადი თვისების გამოყენებით.

დელეგატი

9.3.1. დელეგატს დაუმატეთ 2 ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით. შემდეგ, დელეგატს გამოაკელით I ფუნქციის მისამართი და დაუმატეთ III ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით.

9.3.2. დელეგატს დაუმატეთ 2 ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით. შემდეგ, დელეგატს გამოაკელით II ფუნქციის მისამართი და დაუმატეთ III ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით.

9.3.3. დელეგატს დაუმატეთ 3 ფუნქციის მისამართი. სამივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით. შემდეგ, დელეგატს გამოაკელით I და III ფუნქციების მისამართები და დაუმატეთ IV ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით.

9.3.4. დელეგატს დაუმატეთ 3 ფუნქციის მისამართი. სამივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით. შემდეგ, დელეგატს გამოაკელით II და III ფუნქციების მისამართები და დაუმატეთ IV ფუნქციის მისამართი. ორივე ფუნქცია გამოიძახეთ დელეგატის საშუალებით.

მოვლენა

9.4.1. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც პირველი რიცხვი მეორეზე მეტია. დამამუშავებელს გამოაქვს შესაბამისი შეტყობინება.

9.4.2. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც ორივე რიცხვი ლუწია. დამამუშავებელს გამოაქვს შესაბამისი შეტყობინება.

9.4.3. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც ორი სტრიქონი ერთნაირი არ არის.

9.4.4. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც მასივის ყველა ელემენტი 50-ზე ნაკლებია.

9.4.5. შექმნენით მოვლენა, რომელიც აღიძვრება ნულზე გაყოფის შემთხვევაში.

9.4.6. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც 5-ელემენტიან მთელრიცხვა მასივში მხოლოდ ნულებია.

9.4.7. შექმენით მოვლენა, რომელიც აღიძვრება მაშინ, როდესაც 25-სიმბოლოიან სტრიქონში მხოლოდ “*”-ებია.

სახელების სივრცე

9.5.1. შექმენით სახელების 2 სივრცე - Sivrce_1 და Sivrce_2 რომლებიც შეიცავენ Samkutxedi კლასს. Sivrce_1 სივრცის Samkutxedi კლასი შეიცავს Fartobi ფუნქციას, რომელიც სამკუთხედის ფართობს ანგარიშობს, Sivrce_2 სივრცის Samkutxedi კლასი კი შეიცავს Perimetri ფუნქციას, რომელიც სამკუთხედის პერიმეტრს ანგარიშობს.

9.5.2. შექმენით სახელების 2 სივრცე - Sivrce_1 და Sivrce_2 რომლებიც შეიცავენ Otxkutxedi კლასს. Sivrce_1 სივრცის Otxkutxedi კლასი შეიცავს Fartobi ფუნქციას, რომელიც ოთხკუთხედის ფართობს ანგარიშობს, Sivrce_2 სივრცის Otxkutxedi კლასი კი შეიცავს Perimetri ფუნქციას, რომელიც ოთხკუთხედის პერიმეტრს ანგარიშობს.

9.5.3. შექმენით სახელების 2 სივრცე - Sivrce_1 და Sivrce_2 რომლებიც შეიცავენ Masivi კლასს.

Sivrce_1 სივრცის Masivi კლასი შეიცავს Funqcia1 ფუნქციას, რომელიც გამოთვლის და დააბრუნებს ერთგანზომილებიანი მასივის დადებითი ელემენტების ჯამს, Sivrce_2 სივრცის Masivi კლასი კი შეიცავს Funqcia2 ფუნქციას, რომელიც გამოთვლის და დააბრუნებს ერთგანზომილებიანი მასივის უარყოფითი ელემენტების ნამრავლს.

ინტერფეისი

9.6.1. შექმენით ინტერფეისი, რომელიც ორ ფუნქციას შეიცავს. I ფუნქცია გამოთვლის და აბრუნებს პარამეტრის კვადრატს, II კი - პარამეტრის კუბს. მოახდინეთ ამ ინტერფეისის რეალიზება.
9.6.2. შექმენით ინტერფეისი, რომელიც ერთ თვისებას შეიცავს. თვისება პრივატულ ცვლადს ლუწ მნიშვნელობებს ანიჭებს. მოახდინეთ ამ ინტერფეისის რეალიზება.
9.6.3. შექმენით ინტერფეისი, რომელიც ერთ ინდექსირებულ თვისებას შეიცავს. თვისება პრივატულ ცვლადს 20-ზე მეტ მნიშვნელობებს ანიჭებს. მოახდინეთ ამ ინტერფეისის რეალიზება.
9.6.4. შექმენით ინტერფეისი, რომელიც ერთ მოვლენას შეიცავს. მოვლენა აღიძვრება მაშინ, როდესაც შეტანილი რიცხვი 25-ს გადააჭარბებს. მოახდინეთ ამ ინტერფეისის რეალიზება.
9.6.5. შექმენით ინტერფეისი, რომელიც ერთ ფუნქციას შეიცავს. ფუნქცია გამოთვლის და აბრუნებს პარამეტრის კუბს; შეიცავს ერთ თვისებას, რომელიც პრივატულ ცვლადს დადებით მნიშვნელობებს ანიჭებს; შეიცავს ერთ ინდექსატორს, რომელიც 4 მთელ რიცხვთან მუშაობს და ანიჭებს მათ 7-ის ჯერად მნიშვნელობებს; შეიცავს ერთ მოვლენას, რომელიც აღიძვრება მაშინ, როდესაც ერთგანზომილებიანი მასივის მინიმალური ელემენტი იქნება 10-ზე ნაკლები.

თავი 10. ინფორმაციის შეტანა-გამოტანა

ფაილში ბაიტების შეტანა-გამოტანა. FileStream კლასი

შეადგინეთ პროგრამა, რომელიც:
10.1.1. ერთი ფაილიდან წაიკითხავს byte ტიპის 5 მთელ რიცხვს, მოათავსებს მათ ერთგანზომილებიან მასივში და ამ მასივის ელემენტების ჯამს დაუმატებს ამავე ფაილს.
10.1.2. ერთი ფაილიდან წაიკითხავს byte ტიპის 5 მთელ რიცხვს, მოათავსებს მათ ერთგანზომილებიან მასივში და ამ მასივის ელემენტებს და მათ ჯამს ჩაწერს მეორე ფაილში.
10.1.3. ერთი ფაილიდან მეორეში სათითაოდ გადაწერს byte ტიპის 10 მთელ რიცხვს.

ფაილში სიმბოლოების შეტანა-გამოტანა. StreamReader და StreamWriter კლასები

შეადგინეთ პროგრამა, რომელიც:
10.2.1. ერთი ფაილიდან წაიკითხავს 10 სიმბოლოს, მოათავსებს მათ სიმბოლოების მასივში და მასივის ელემენტებს ჩაწერს მეორე ფაილში.
10.2.2. ერთი ფაილიდან წაიკითხავს 10 სიმბოლოს, მოათავსებს მათ სიმბოლოების მასივში და ამ მასივის ელემენტებს დაუმატებს ამავე ფაილს.
10.2.3. ერთი ფაილიდან წაიკითხავს სტრიქონს და ჩაწერს მას მეორე ფაილში.
10.2.4. ერთი ფაილიდან მეორეში სათითაოდ გადაწერს char ტიპის 10 სიმბოლოს.

ფაილში ორობითი მონაცემების შეტანა-გამოტანა. BinaryReader და BinaryWriter ნაკადები

შეადგინეთ პროგრამა, რომელიც:
10.3.1. ფაილიდან წაიკითხავს 10 მთელ რიცხვს და ჩაწერს მათ ფაილში.
10.3.2. ფაილიდან წაიკითხავს 10 წილად რიცხვს და ჩაწერს მათ ფაილში.
10.3.3. ფაილიდან წაიკითხავს 20 სიმბოლოიან სტრიქონს და ჩაწერს მათ ფაილში.

10.3.4. ფაილიდან წაიკითხავს 10 მთელ რიცხვს და დაუმატებს მათ ამავე ფაილს.
10.3.5. ფაილიდან წაიკითხავს 10 წილად რიცხვს და დაუმატებს მათ ამავე ფაილს.
10.3.6. ფაილიდან წაიკითხავს 15 მთელ რიცხვს და მოათავსებს მათ ერთგანზომილებიან მასივში.

ფაილებთან პირდაპირი მიმართვა. Seek ფუნქცია

შეადგინეთ პროგრამა, რომელიც:
10.4.1. ფაილის მე-2 პოზიციიდან წაიკითხავს 10 ბაიტს და მოათავსებს მათ ერთგანზომილებიან მასივში.
10.4.2. ფაილის მე-5 პოზიციიდან ჩაწერს 4 ბაიტს ერთგანზომილებიანი მასივიდან.
10.4.3. ფაილის მე-3 პოზიციიდან ჩაწერს 4 სიმბოლოს.
10.4.4. ფაილის მე-6 პოზიციიდან წაიკითხავს 4 სიმბოლოს.

ფაილის შესახებ ინფორმაციის მიღება. FileInfo კლასი

10.5.1. მიიღეთ მითითებული ფაილის ატრიბუტები.
10.5.2. მიიღეთ მითითებული ფაილის შექმნის თარიღი.
10.5.3. მიიღეთ იმ კატალოგის სახელი, რომელშიც მითითებული ფაილია მოთავსებული.
10.5.4. შეამოწმეთ მითითებული ფაილი არსებობს თუ არა.
10.5.5. მიიღეთ მითითებული ფაილის გაფართოება.
10.5.6. მიიღეთ მითითებული ფაილის სრული სახელი.
10.5.7. მიიღეთ მითითებულ ფაილთან უკანასკნელი მიმართვის თარიღი.
10.5.8. მიიღეთ მითითებულ ფაილში უკანასკნელი ჩაწერის თარიღი.
10.5.9. მიიღეთ მითითებული ფაილის ზომა.

კატალოგებთან მუშაობა. DirectoryInfo კლასი

10.6.1. მიიღეთ მითითებული კატალოგის ატრიბუტები.
10.6.2. მიიღეთ მითითებული კატალოგის შექმნის თარიღი.
10.6.3. მიიღეთ მითითებული კატალოგის სრული სახელი.
10.6.4. მიიღეთ მითითებულ კატალოგთან უკანასკნელი მიმართვის დრო.
10.6.5. მიიღეთ მითითებულ კატალოგში უკანასკნელი ჩაწერის დრო.
10.6.6. შეამოწმეთ მითითებული კატალოგის არსებობა.
10.6.7. მიიღეთ მითითებული კატალოგის მშობელი კატალოგის სახელი.
10.6.8. მიიღეთ მითითებული კატალოგის ძირითადი კატალოგის სახელი.
10.6.9. შექმენით კატალოგი.
10.6.10. შექმენით ქვეკატალოგი.
10.6.11. წაშალეთ კატალოგი.
10.6.12. მიიღეთ მიმდინარე კატალოგის ქვეკატალოგების სია.
10.6.13. მიიღეთ მიმდინარე კატალოგის ფაილების სია.
10.6.14. ქვეკატალოგი გადაიტანეთ ერთი კატალოგიდან მეორეში.

სხვადასხვა

შეადგინეთ პროგრამა, რომელიც:
10.7.1. ფაილიდან წაიკითხავს მთელ რიცხვებს და დათვლის მათ შორის უარყოფითი, ნულოვანი და დადებითი რიცხვების რაოდენობას.
10.7.2. ფაილიდან წაიკითხავს წილად რიცხვებს, დათვლის მათ რაოდენობას და გამოთვლის მათ ჯამს.
10.7.3. წაიკითხავს ფაილში ჩაწერილი წილადი რიცხვების შუაში მოთავსებულ რიცხვს. ფაილში

ჩაწერილია კენტი რაოდენობის რიცხვი.
10.7.4. ფაილიდან წაიკითხავს მთელ რიცხვებს, დაალაგებს მათ ზრდადობის მიხედვით და ჩაწერს ამავე ფაილში.
10.7.5. ფაილიდან წაიკითხავს რიცხვებს და მათ შორის პირველსა და უკანასკნელს მეორე ფაილში გადაწერს.
10.7.6. ფაილიდან წაიკითხავს რიცხვებს და მეორე ფაილში გადაწერს:
ა. მათ შორის მინიმალურსა და მაქსიმალურს;
ბ. კენტი ინდექსის მქონე რიცხვებს შორის უდიდესს;
გ. ლუწი ინდექსის მქონე რიცხვებს შორის უმცირესს.
დ. ლუწი და კენტი რიცხვების რაოდენობას;
10.7.7. სიმბოლური ფაილიდან ორ მონაცემს კითხულობს. თუ ეს მონაცემები ციფრებია, მაშინ მათი ჯამი, ნამრავლი და სხვაობა ამავე ფაილს დაუმატეთ.
10.7.8. ტექსტური ფაილის თითოეულ სტრიქონში დათვლის სიტყვების რაოდენობას.
10.7.9. ფაილიდან წაიკითხავს რიცხვებს და მეორე ფაილში გადაწერს:
ა. კენტ რიცხვებს;
ბ. ლუწ რიცხვებს;
გ. 7-ის ჯერად რიცხვებს;
დ. რიცხვებს, რომლებიც წარმოადგენენ კვადრატებს.
10.7.10. განსაზღვრავს ორი ფაილი შეიცავს თუ არა ერთნაირ რიცხვებს.
10.7.11. ერთი სიმბოლური ფაილიდან მეორეში მონაცემებს ისე გადაწერს, რომ მიმდევრობით მოთავსებული რამდენიმე ინტერვალი შეიცვალოს ერთით.
10.7.12. ფაილის ყოველი სტრიქონის წინ ინტერვალის ნიშანს ჩასვამს და გადაწერს მეორე ფაილში.
10.7.13. ფაილის ყოველი სტრიქონის წინ და ბოლოში მოთავსებულ ინტერვალებს წაშლის და მიღებულ სტრიქონებს ამავე ფაილში ჩაწერს.
10.7.14. სიმბოლურ ფაილში:
ა. დათვლის 'ა' სიმბოლოს რაოდენობას;
ბ. დათვლის 'ს', 'რ' და 'ე' სიმბოლოების რაოდენობას;
გ. დაადგენს შეიცავს თუ არა ფაილი "კომპიუტერი" სიმბოლოების მიმდევრობას;
დ. დათვლის "კომპიუტერი" სიტყვის რაოდენობას.
10.7.15. სიმბოლური ფაილიდან წაშლის ყველა სამსიმბოლოიან სიტყვას და შედეგს მეორე ფაილში გადაწერს.
10.7.16. სიმბოლურ ფაილში:
ა. იპოვის ყველაზე გრძელ სიტყვას;
ბ. იპოვის ყველაზე მოკლე სიტყვას;
გ. იპოვის ყველაზე გრძელ სიტყვას, რომლის მესამე სიმბოლოა 'ფ';
დ. დათვლის ორი, სამი და ოთხი სიმბოლოსაგან შემდგარი სიტყვების რაოდენობას.
10.7.17. ერთ სიმბოლურ ფაილს მეორეში გადაწერს.
10.7.18. ერთი სიმბოლური ფაილიდან წაიკითხავს ტექსტს, პატარა ასოებს დიდ ასოებად გარდაქმნის და შედეგს მეორე ფაილში ჩაწერს.
10.7.19. ერთი სიმბოლური ფაილიდან მონაცემებს უკუ მიმდევრობით გადაწერს მეორე ფაილში.
10.7.20. ტექსტურ ფაილში მოძებნის 9 სიტყვას, რომლებიც ყველაზე ხშირად გვხვდება.
10.7.21. ტექსტურ ფაილში დათვლის სტრიქონების რაოდენობას.
10.7.22. ტექსტური ფაილის თითოეულ სტრიქონში დათვლის სიმბოლოების რაოდენობას.
10.7.23. ეკრანზე გამოიტანს ტექსტური ფაილის ყველაზე გრძელ და მოკლე სტრიქონებს.
10.7.24. ტექსტურ ფაილში დათვლის იმ სტრიქონების რაოდენობას, რომლებშიც სიმბოლოების რაოდენობა 60-ს აღემატება.

10.7.25. ფაილში მოთავსებულ ტექსტს ეკრანზე გამოიტანს.
10.7.26. ფაილში ჩაწერს 9 სტუდენტის გვარს, დაბადების თარიღსა და სიმაღლეს.
10.7.27. ფაილის მეხუთე პოზიციიდან 3 ძახილის ნიშანს ჩაწერს.
10.7.28. ერთი ფაილის კენტი ნომრის მქონე სიმბოლოებს მეორე ფაილის ლუწი ნომრის პოზიციებში გადაწერს.
10.7.29. ერთი ფაილიდან სიმბოლოებს დაწყებული პირველი პოზიციიდან გადაწერს მეორე ფაილში დაწყებული უკანასკნელი პოზიციიდნ.
10.7.30. მოცემულია ორი ფაილი, რომლებიც შეიცავენ ზრდადობით დალაგებულ რიცხვებს. შეადგინეთ პროგრამა, რომელიც მესამე ფაილს შექმნის და მასში ზრდადობის მიხედვით ჩაწერს ორივე ფაილში მოთავსებულ რიცხვებს.
10.7.31. მოცემულია ორი ტექსტური ფაილი. შეადგინეთ პროგრამა, რომელიც მესამე ფაილში ჯერ ჩაწერს პირველ ფაილში მოთავსებულ ტექსტს, შემდეგ კი - მეორე ფაილში მოთავსებულ ტექსტს.
10.7.32. ტექსტური ფაილი შეიცავს C++ ენაზე შედგენილ საწყის კოდს (პროგრამას). კოდის თითოეული ოპერატორი თითო სტრიქონს იკავებს. შეადგინეთ პროგრამა, რომელიც:
ა. შეამოწმებს გამხსნელი და დამხურავი მრგვალი ფრჩხილების გამოყენების სისწორეს;
ბ. შეამოწმებს გამხსნელი და დამხურავი ფიგურული ფრჩხილების გამოყენების სისწორეს;

თავი 11. განსაკუთრებული სიტუაცია

განსაკუთრებული სიტუაცია. try, catch და throw ოპერატორები

11.1.1. შეადგინეთ პროგრამა, რომელიც შეკრებს ერთგანზომილებიანი მასივის ელემენტებს. ინდექსის არასწორი მნიშვნელობის შემთხვევაში გამოიტანეთ შესაბამისი შეტყობინება.
11.1.2. შეადგინეთ პროგრამა, რომელიც შეკრებს ორგანზომილებიანი მასივის ელემენტებს. რომელიმე ინდექსის არასწორი მნიშვნელობის შემთხვევაში გამოიტანეთ შესაბამისი შეტყობინება.
11.1.3. შეადგინეთ პროგრამა, რომელიც ერთი ერთგანზომილებიანი მასივის ელემენტებს გაყოფს მეორე ერთგანზომილებიანი მასივის ელემენტებზე. ნულზე გაყოფის შემთხვევაში გამოიტანეთ შესაბამისი შეტყობინება.
11.1.4. შეადგინეთ პროგრამა, რომელიც მთელრიცხვა ერთგანზომილებიანი მასივის ელემენტებს ფაილში ჩაწერს. შეტანა-გამოტანის შეცდომის აღძვრის შემთხვევაში გამოიტანეთ შესაბამისი შეტყობინება.
11.1.5. შეადგინეთ პროგრამა, რომელიც მთელრიცხვა ორგანზომილებიანი მასივის ელემენტებს ფაილიდან წაიკითხავს. შეტანა-გამოტანის შეცდომის აღძვრის შემთხვევაში გამოიტანეთ შესაბამისი შეტყობინება.

თავი 12. სტრიქონი

სტრიქონი

ქვემოთ მოცემულ ამოცანებში სტრიქონების შესატანად გამოიყენეთ textBox კომპონენტი. შეადგინეთ პროგრამა, რომელიც:
12.1.1. სტრიქონში დათვლის სიმბოლოების რაოდენობას.
12.1.2. სტრიქონში დათვლის "ბ" სიმბოლოს რაოდენობას.
12.1.3. სტრიქონში დათვლის ციფრების რაოდენობას 0-დან 9-მდე.
12.1.4. სტრიქონში დათვლის სასვენი ნიშნების რაოდენობას (, ; ! ? :).

12.1.5. სტრიქონში დათვლის ხმოვნების რაოდენობას (ა, ი, ო, უ, ე).
12.1.6. სტრიქონში იპოვის "ა" სიმბოლოს პოზოციას.
12.1.7. სტრიქონში ყველა "ი" სიმბოლოს "ს" სიმბოლოთი შეცვლის.
12.1.8. სტრიქონში ყველა ციფრს "რ" სიმბოლოთი შეცვლის.
12.1.9. სტრიქონში "ეს არის ტესტი" ჩაუმატებს "კომპიუტერი" სიტყვას დაწყებული მე-3 პოზიციიდან.
12.1.10. სტრიქონიდან "ეს არის ტესტი" წაშლის "არის" სიტყვას.
12.1.11. სტრიქონიდან "ეს არის ტესტი" წაშლის 6 სიმბოლოს დაწყებული მე-2 პოზიციიდან.
12.1.12. სტრიქონში "ეს არის ტესტი" "ეს" სიტყვას შეცვლის "კომპიუტერი" სიტყვით.
12.1.13. სტრიქონში "ეს არის ტესტური პროგრამა" "ეს არის" სიტყვებს შეცვლის "ახლა შევასრულოთ" სიტყვებით.

დინამიკური სტრიქონი

12.2.1. შექმენით დინამიკური სტრიქონი. კონსტრუქტორს გადაეცით თქვენი სახელი და გვარი. დინამიკური სტრიქონის ტევადობაა 50. დინამიკურ სტრიქონს დაუმატეთ წილადი.
12.2.2. შექმენით დინამიკური სტრიქონი, რომლის ტევადობაა 50, მაქსიმალური ტევადობა კი - 70. დინამიკურ სტრიქონს დაუმატეთ თქვენი სახელი, გვარი და მთელი რიცხვი.
12.2.3. შექმენით დინამიკური სტრიქონი. კონსტრუქტორს გადაეცით თქვენი სახელი და გვარი. დინამიკურ სტრიქონს დაუმატეთ ლოგიკური მნიშვნელობა.
12.2.4. შექმენით დინამიკური სტრიქონი. კონსტრუქტორს გადაეცით თქვენი სახელი და გვარი, საწყისი ინდექსი, გადასაწერი სიმბოლოების რაოდენობა და ტევადობა. საწყისი ინდექსია 3, გადასაწერი სიმბოლოების რაოდენობაა 7, ტევადობაა 50. დინამიკურ სტრიქონს მე-4 პოზიციიდან ჩაუმატეთ სიტყვა "კომპიუტერი".

სხვადასხვა

შეადგინეთ პროგრამა, რომელიც:
12.3.1. სიმბოლოების $M_{5,5}$ მასივში მოძებნის იმ სტრიქონს, რომელიც შეიცავს "ა" სიმბოლოს მაქსიმალურ რაოდენობას.
12.3.2. სიმბოლოების $M_{5,5}$ მასივში მოძებნის იმ სტრიქონს, რომელიც შეიცავს განსხვავებული სიმბოლოების მაქსიმალურ რაოდენობას.
12.3.3. სიმბოლოების $M_{5,5}$ მასივში მოძებნის იმ სვეტს, რომელიც შეიცავს "ა" სიმბოლოს მინიმალურ რაოდენობას.
12.3.4. სიმბოლოების $M_{5,5}$ მასივში მოძებნის იმ სვეტს, რომელიც შეიცავს განსხვავებული სიმბოლოების მინიმალურ რაოდენობას.
12.3.5. სტრიქონის მე-10 პოზიციიდან 30-ე პოზიციამდე დათვლის "+" სიმბოლოს რაოდენობას.
12.3.6. სტრიქონში დათვლის "რო" სიმბოლოების რაოდენობას.
12.3.7. სტრიქონში დათვლის ინტერვალების რაოდენობას.
12.3.8. განსაზღვრავს შეიცავს თუ არა სტრიქონი "დ" სიმბოლოს.
12.3.9. განსაზღვრავს შეიცავს თუ არა სტრიქონი ერთნაირი სიმბოლოების წყვილებს.
12.3.10. განსაზღვრავს შეიცავს თუ არა სტრიქონი 4 მეზობელ "კ" სიმბოლოს.
12.3.11. სტრიქონში განსაზღვრავს მეზობელი ინტერვალების ყველაზე გრძელი მიმდევრობის ზომას.
12.3.12. სტრიქონიდან წაშლის "ათი" სიტყვებს.
12.3.13. სტრიქონში "ს" სიმბოლოს შემდეგ მოთავსებულ ყველა სიმბოლოს "თ" სიმბოლოთი შეცვლის.
12.3.14. ინგლისური ანბანის ყველა ასოს გამოიტანს.
12.3.15. ქართული ანბანის ყველა ასოს გამოიტანს.

12.3.16. კლავიატურის ყველა სიმბოლოს გამოიტანს.
12.3.17. სტრიქონში დათვლის სიტყვების რაოდენობას.
12.3.18. სტრიქონის უკანასკნელ სიტყვაში "ლ" სიმბოლოს რაოდენობას დათვლის.
12.3.19. სტრიქონში იმ სიტყვების რაოდენობას დათვლის, რომლებიც "ო" სიმბოლოთი იწყება.
12.3.20. სტრიქონში იმ სიტყვების რაოდენობას დათვლის, რომლებიც "შვილი" სიმბოლოებით მთავრდება.
12.3.21. სტრიქონში იმ სიტყვების რაოდენობას დათვლის, რომელთა პირველი და უკანასკნელი ასოები ერთმანეთს ემთხვევა.
12.3.22. სტრიქონში ყველაზე გრძელ სიტყვას იპოვის.
12.3.23. სტრიქონში ყველაზე მოკლე სიტყვას იპოვის.
12.3.24. სტრიქონიდან წაშლის ციფრებს.
12.3.25. სტრიქონში პატარა ასოებს დიდი ასოებით შეცვლის.
12.3.26. სტრიქონში დიდ ასოებს პატარა ასოებით შეცვლის.
12.3.27. სტრიქონში წაშლის წერტილის წინ მოთავსებულ რიცხვებს.
12.3.28. სტრიქონში წაშლის წერტილის შემდეგ მოთავსებულ რიცხვებს.
12.3.29. სტრიქონის სიმბოლოებს უკუმიმდევრობით დაალაგებს.
12.3.30. ტექსტში მრგვალი ფრჩხილების ბალანსს შეამოწმებს. მრგვალი ფრჩხილები დაბალანსებულია თუ გამხსნელ ფრჩხილს მოსდევს შესაბამისი დამხურავი ფრჩხილი და მათი რაოდენობა ტოლია.
12.3.31. შეამოწმებს არის თუ არა ერთი წინადადება მეორის ნაწილი და პირიქით.
12.3.32. შეამოწმებს ერთნაირია თუ არა ორი წინადადება.
12.3.33. ორ წინადადებაში მოძებნის იმ სიტყვებს, რომლებიც ორივე წინადადებაში შედიან და ამ სიტყვებს ჩაწერს სტრიქონების მასივში.
12.3.34. მოცემულ რიცხვს (1-დან 10-მდე) ჩაწერს სიტყვების საშუალებით.
12.3.35. სტრიქონში მოძებნის იმ სიტყვებს, რომლებიც "ძე" სიმბოლოებით მთავრდება და ამ სიტყვებს ჩაწერს სტრიქონების მასივში.
12.3.36. განსაზღვრავს ტექსტში რამდენი სიტყვა შეიცავს 1 ხმოვანს, 2 ხმოვანს, 3 ხმოვანს და ა.შ.
12.3.37. განსაზღვრავს ტექსტში სიტყვების რამდენი პროცენტი იწყება "ლ" სიმბოლოთი.
12.3.38. განსაზღვრავს შეიცავს თუ არა ტექსტი ასოებისაგან განსხვავებულ სიმბოლოებს.
12.3.39. ტექსტში იპოვის იმ სიტყვას, რომელიც შეიცავს ხმოვნების მაქსიმალურ რაოდენობას.
12.3.40. ტექსტში იპოვის იმ სიტყვას, რომელიც შეიცავს თანხმოვნების მინიმალურ რაოდენობას.
12.3.41. ტექსტში იპოვის იმ სიტყვას, რომელიც შეიცავს "გ" სიმბოლოს მაქსიმალურ რაოდენობას.
12.3.42. ტექსტში იპოვის სიტყვას, რომელიც ყველაზე ხშირად გვხვდება.
12.3.43. ტექსტში იპოვის 10 სიტყვას, რომელიც ყველაზე ხშირად გვხვდება და დათვლის თითოეული მათგანის რაოდენობას.
12.3.44. ტექსტში განსაზღვრავს თუ რომელი ასოთი იწყება სიტყვების უმრავლესობა.
12.3.45. ტექსტში განსაზღვრავს თუ რამდენჯერ გვხვდება ანბანის თითოეული ასო.
12.3.46. ტექსტში განსაზღვრავს სიტყვების პირველი ასოების მინიმალურ რაოდენობას, რომელთა მიხედვით შეიძლება განვასხვავოთ სიტყვები.
12.3.47. ტექსტში იპოვის იმ სიტყვებს, რომლებიც შეიცავენ გაორმაგებულ ხმოვნებს.
12.3.48. ტექსტში იპოვის იმ სიტყვებს, რომლებიც შეიცავენ გაორმაგებულ თანხმოვნებს.
12.3.49. ტექსტიდან წაშლის ზედმეტ ინტერვალის ნიშნებს და სიტყვებს შორის დატოვებს ინტერვალის თითო ნიშანს.
12.3.50. მოცემულია ორი სიტყვა. ისინი გააერთიანეთ ისე, რომ წინ მოთავსთდეს მოკლე სიტყვა, შემდეგ კი - გრძელი.
12.3.51. მოცემულია ორი სიტყვა. მეორე სიტყვა მოათავსეთ პირველი სიტყვის შუაში.

12.3.52. დაადგინეთ რამდენი პალინდრომია მოცემულ ტექსტში. პალინდრომია, მაგალითად, "აბგგბა", 123321 და ა.შ.

თავი 14. კოლექცია

დინამიკური მასივი, ჰეშ-ცხრილი, დახარისხებული სია

14.1.1. შექმენით 10-ელემენტიანი დინამიკური მასივი სტრიქონების მასივის საფუძველზე. დინამიკურ მასივს დაუმატეთ ორი სტრიქონი. ეკრანზე გამოიტანეთ დინამიკურ მასივში ელემენტების რაოდენობა და თვით დინამიკური მასივი.

14.1.2. შექმენით 10-ელემენტიანი დინამიკური მასივი სტრიქონების მასივის საფუძველზე. დინამიკური მასივიდან წაშალეთ სამი სტრიქონი. ეკრანზე გამოიტანეთ დინამიკურ მასივში ელემენტების რაოდენობა და თვით დინამიკური მასივი.

14.1.3. შექმენით 10-ელემენტიანი დინამიკური მასივი სტრიქონების მასივის საფუძველზე. დინამიკურ მასივს ჩაუმატეთ სტრიქონი მე-2 პოზიციიდან. ეკრანზე გამოიტანეთ დინამიკურ მასივში ელემენტების რაოდენობა და თვით დინამიკური მასივი.

14.1.4. შექმენით 10-ელემენტიანი დინამიკური მასივი სტრიქონების მასივის საფუძველზე. დინამიკურ მასივში მოძებნეთ თქვენი სახელი. ეკრანზე გამოიტანეთ მოძებნილი ელემენტის ინდექსი, დინამიკურ მასივში ელემენტების რაოდენობა და თვით დინამიკური მასივი.

14.1.5. შექმენით 10-ელემენტიანი ჰეშ-ცხრილი. ჰეშ-ცხრილს დაუმატეთ ორი ელემენტი. ეკრანზე გამოიტანეთ ჰეშ-ცხრილში ელემენტების რაოდენობა და ყველა გასაღები და ყველა სიდიდე.

14.1.6. შექმენით 10-ელემენტიანი ჰეშ-ცხრილი. ჰეშ-ცხრილიდან წაშალეთ ორი ელემენტი. ეკრანზე გამოიტანეთ ჰეშ-ცხრილში ელემენტების რაოდენობა და ყველა გასაღები და ყველა სიდიდე.

14.1.7. შექმენით 10-ელემენტიანი ჰეშ-ცხრილი. ჰეშ-ცხრილში შეასრულეთ ძებნა გასაღებისა და სიდიდის მიხედვით. ნაპოვნი გასაღები და სიდიდე ეკრანზე გამოიტანეთ. ეკრანზე გამოიტანეთ ჰეშ-ცხრილში ელემენტების რაოდენობა და ყველა გასაღები და ყველა სიდიდე.

14.1.8. შეკუმშეთ 20-ელემენტიანი დინამიკური მასივი მისი გამეორებადი ელემენტების წაშლის გზით.

14.1.9. შეკუმშეთ 20-ელემენტიანი დინამიკური მასივი მისი იმ ელემენტების წაშლის გზით, რომლებიც მასივში ერთხელ გვხვდება.

14.1.10. შეკუმშეთ 20-ელემენტიანი დინამიკური მასივი მისი ნულოვანი ელემენტების წაშლის გზით.

რიგი, სტეკი და ბიტების მასივი

14.2.1. შექმენით 10-ელემენტიანი რიგი. წაიკითხეთ და ეკრანზე გამოიტანეთ რიგის I ელემენტის მნიშვნელობა მისი წაშლის გარეშე. ეკრანზე გამოიტანეთ რიგში არსებული ელემენტების რაოდენობა და რიგის ყველა ელემენტი. რიგიდან ყველა ელემენტი წაშალეთ.

14.2.2. შექმენით 10-ელემენტიანი რიგი. წაიკითხეთ და ეკრანზე გამოიტანეთ რიგის I ელემენტის მნიშვნელობა მისი წაშლის გარეშე. ეკრანზე გამოიტანეთ რიგში არსებული ელემენტების რაოდენობა და რიგის ყველა ელემენტი. რიგიდან ყველა ელემენტი წაშალეთ.

14.2.3. შექმენით 10-ელემენტიანი სტეკი. წაიკითხეთ და ეკრანზე გამოიტანეთ სტეკის უკანასკნელი ელემენტის მნიშვნელობა მისი წაშლის გარეშე. ეკრანზე გამოიტანეთ სტეკში არსებული ელემენტების რაოდენობა და სტეკის ყველა ელემენტი. სტეკიდან ყველა ელემენტი წაშალეთ.

14.2.4. შექმენით 10-ელემენტიანი სტეკი. წაიკითხეთ და ეკრანზე გამოიტანეთ სტეკის

უკანასკნელი ელემენტის მნიშვნელობა მისი წაშლის გარეშე. ეკრანზე გამოიტანეთ სტეკში არსებული ელემენტების რაოდენობა და სტეკის ყველა ელემენტი. სტეკიდან ყველა ელემენტი წაშალეთ.

14.2.5. შექმენით ბიტების 2 მასივი, თითოეული 10-ელემენტიანი. ამ მასივებზე შეასრულეთ **და, ან, არა** და **გამომრიცხავი ან** ოპერაციები.

თავი 15. შესრულების ნაკადი

შესრულების ნაკადი

15.1.1. შექმენით 2 ნაკადი. პირველ ნაკადში მუშაობს ფუნქცია, რომელიც ანგარიშობს სამკუთხედის პერიმეტრს. მეორე ნაკადში მუშაობს ფუნქცია, რომელიც ანგარიშობს სამკუთხედის ფართობს.

15.1.2. შექმენით 3 ნაკადი. პირველ ნაკადში მუშაობს ფუნქცია, რომელიც ანგარიშობს სამკუთხედის პერიმეტრს. მეორე ნაკადში მუშაობს ფუნქცია, რომელიც ანგარიშობს სამკუთხედის ფართობს. მესამე ნაკადში მუშაობს ფუნქცია, რომელიც გამოთვლის ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტების ჯამს.

15.1.3. შექმენით ფუნქცია, რომელიც ახდენს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტების შეკრებას. ეს ფუნქცია ორ ნაკადში შეასრულეთ.

15.1.4. შექმენით ფუნქცია, რომელიც ახდენს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტების შეკრებას. ეს ფუნქცია სამ ნაკადში შეასრულეთ.

ნაკადის პრიორიტეტი

15.2.1. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. პირველი ნაკადის პრიორიტეტია უდაბლესი, მეორე ნაკადის კი - უმაღლესი.

15.2.2. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია სამ ნაკადად შეასრულეთ. პირველი ნაკადის პრიორიტეტია უდაბლესი, მეორე ნაკადის - ნორმალური, მესამე ნაკადის კი - უმაღლესი.

ნაკადის მდგომარეობა

15.3.1. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადმა დაიწყო თუ არა შესრულება.

15.3.2. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი სრულება თუ არა.

15.3.3. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი სრულება თუ არა ფონურ რეჟიმში.

15.3.4. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი ბლოკირებულია თუ არა Wait, Sleep ან Join ფუნქციის მიერ.

15.3.5. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ მოთხოვნილია თუ არა ნაკადის დროებით შეჩერება.

15.3.6. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი დროებით შეჩერებულია

თუ არა.

15.3.7. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ მოთხოვნილია თუ არა ნაკადის გაჩერება.

15.3.8. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი გაჩერებულია თუ არა.

15.3.9. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ მოთხოვნილია თუ არა ნაკადის გაუქმება.

15.3.10. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. შეამოწმეთ ნაკადი გაუქმებულია თუ არა.

ნაკადების ლოკალური მონაცემები

15.4.1. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. ორივე ნაკადისთვის მეხსიერების ერთი და იგივე უბნები გამოიყენეთ.

15.4.2. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია სამ ნაკადად შეასრულეთ. სამივე ნაკადისთვის მეხსიერების ერთი და იგივე უბნები გამოიყენეთ.

Timer კლასი

15.5.1. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის დადებით ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. ნაკადი გაუშვით 5 წამის შემდეგ და შეასრულეთ ყოველი 3 წამის შემდეგ.

15.5.2. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის დადებით ელემენტებს. ეს ფუნქცია ერთ ნაკადად შეასრულეთ. ნაკადი გაუშვით დაუყოვნებლივ და შეასრულეთ ყოველი 4 წამის შემდეგ.

Join ფუნქცია

15.6.1. შექმენით ფუნქცია, რომელიც შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. პირველი ნაკადი მიაბით მეორეს.

15.6.2. შექმენით ორი ფუნქცია. პირველი მათგანი შეკრებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს, მეორე კი გაამრავლებს ერთგანზომილებიანი მთელრიცხვა მასივის ელემენტებს. ორივე ფუნქცია ორ ნაკადად შეასრულეთ. პირველი ნაკადი მიაბით მეორეს.

Interlocked კლასი

15.8.1. შექმენით ფუნქცია, რომელიც ახდენს მთელი რიცხვის უსაფრთხო ინკრემენტს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. გამოიყენეთ Interlocked სიტყვა.

15.8.2. შექმენით ფუნქცია, რომელიც ახდენს მთელი რიცხვის უსაფრთხო დეკრემენტს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. გამოიყენეთ Interlocked სიტყვა.

15.8.3. შექმენით ფუნქცია, რომელიც უსაფრთხოდ ანიჭებს მნიშვნელობას მთელ რიცხვს. ეს ფუნქცია ორ ნაკადად შეასრულეთ. გამოიყენეთ Interlocked სიტყვა.

Monitor კლასი

15.9.1. შექმენით ფუნქცია, რომელიც მთელ რიცხვებს შეკრებს 1-დან 20-მდე. ეს ფუნქცია ორ ნაკადში შეასრულეთ. მოახდინეთ ორივე ნაკადის მუშაობის სინქრონიზება საერთო ცვლადთან მუშაობისას Monitor კლასის გამოყენებით.

15.9.2. შექმენით ფუნქცია, რომელიც გამოთვლის 7-ის ფაქტორიალს. ეს ფუნქცია ორ ნაკადში შეასრულეთ. მოახდინეთ ორივე ნაკადის მუშაობის სინქრონიზება საერთო ცვლადთან მუშაობისას Monitor კლასის გამოყენებით.

თავი 16. საბაზო ბიბლიოთეკის სხვა კლასები

გლობალიზაცია

16.1.1. შექმენით ქართული კულტურა. ეკრანზე გამოიტანეთ კულტურის ეროვნული და ინგლისური დასახელება, თარიღისა და დროის ფორმატი, რიცხვის ფორმატი, ვალუტის სიმბოლო, გამყოფი სიმბოლო.
16.1.2. შექმენით დანიის კულტურა. ეკრანზე გამოიტანეთ კულტურის ეროვნული და ინგლისური დასახელება, თარიღისა და დროის ფორმატი, რიცხვის ფორმატი, ვალუტის სიმბოლო, გამყოფი სიმბოლო.
16.1.3. შექმენით ლატვიის კულტურა. ეკრანზე გამოიტანეთ კულტურის ეროვნული და ინგლისური დასახელება, თარიღისა და დროის ფორმატი, რიცხვის ფორმატი, ვალუტის სიმბოლო, გამყოფი სიმბოლო.
16.1.4. შექმენით საბერძნეთის კულტურა. ეკრანზე გამოიტანეთ კულტურის ეროვნული და ინგლისური დასახელება, თარიღისა და დროის ფორმატი, რიცხვის ფორმატი, ვალუტის სიმბოლო, გამყოფი სიმბოლო.

მონაცემების დაშიფვრა და გაშიფვრა

16.2.1. ფაილში ჩაწერეთ დაშიფრული სტრიქონი: თქვენი სახელი და გვარი. შემდეგ ამავე ფაილიდან წაიკითხეთ ეს სტრიქონი, გაშიფრეთ იგი და გამოიტანეთ ეკრანზე. გამოიყენეთ სიმეტრიული კრიპტოგრაფია.
16.2.2. ფაილში ჩაწერეთ დაშიფრული სტრიქონი: თქვენი სახელი და გვარი. შემდეგ ამავე ფაილიდან წაიკითხეთ ეს სტრიქონი, გაშიფრეთ იგი და გამოიტანეთ ეკრანზე. გამოიყენეთ ასიმეტრიული კრიპტოგრაფია.
16.2.3. ფაილში ჩაწერეთ დაშიფრული ერთგანზომილებიანი მთელრიცხვა მასივი. შემდეგ ამავე ფაილიდან წაიკითხეთ ეს მასივი, გაშიფრეთ იგი და გამოიტანეთ ეკრანზე. გამოიყენეთ სიმეტრიული კრიპტოგრაფია.
16.2.4. ფაილში ჩაწერეთ დაშიფრული ერთგანზომილებიანი მთელრიცხვა მასივი. შემდეგ ამავე ფაილიდან წაიკითხეთ ეს მასივი, გაშიფრეთ იგი და გამოიტანეთ ეკრანზე. გამოიყენეთ ასიმეტრიული კრიპტოგრაფია.

სხვადასხვა

კომბინატორიკა

1.1. შეადგინეთ პროგრამა, რომელიც დააფორმირებს ხუთკაციანი გუნდის ყველა შესაძლო ვარიანტს არსებული ცხრა კანდიდატურისაგან.
1.2. შეადგინეთ პროგრამა, რომელიც მოძებნის ისეთ ოთხნიშნა რიცხვებს, რომელთა ციფრების ჯამი წინასწარ მოცემული N რიცხვის ტოლია.
1.3. შეადგინეთ პროგრამა, რომელიც დააფორმირებს 1,2,3,4 ციფრების ყველა შესაძლო გადაადგილებას.
1.4. შეადგინეთ პროგრამა, რომელიც 1,2,3,4,5,6,7 ციფრებისაგან დააფორმირებს ყველა შესაძლო ოთხთანრიგა რიცხვს.

დახარისხება

2.1. შეადგინეთ პროგრამა, რომელიც ერთგანზომილებიანი მასივის ელემენტებს ზრდადობის მიხედვით დაალაგებს.
2.2. შეადგინეთ პროგრამა, რომელიც ერთგანზომილებიანი მასივის ელემენტებს კლებადობის მიხედვით დაალაგებს.
2.3. შეადგინეთ პროგრამა, რომელიც ორგანზომილებიანი მასივის თითოეული სტრიქონის ელემენტებს კლებადობის მიხედვით დაალაგებს.
2.4. შეადგინეთ პროგრამა, რომელიც ორგანზომილებიანი მასივის თითოეული სვეტის ელემენტებს ზრდადობის მიხედვით დაალაგებს.
2.5. შეადგინეთ პროგრამა, რომელიც შეამოწმებს ერთგანზომილებიანი მასივი არის თუ არა ზრდადობის მიხედვით დალაგებული.
2.6. შეადგინეთ პროგრამა, რომელიც შეამოწმებს ერთგანზომილებიანი მასივი არის თუ არა კლებადობის მიხედვით დალაგებული.
2.7. მასივი დალაგებულია ზრდადობის მიხედვით. შეადგინეთ პროგრამა, რომელიც ამ მასივში ჩასვამს R რიცხვს მასივის ზრდადობის მიხედვით მოწესრიგების დაურღვევლად.
2.8. მასივი დალაგებულია კლებადობის მიხედვით. შეადგინეთ პროგრამა, რომელიც ამ მასივში ჩასვამს R რიცხვს მასივის კლებადობის მიხედვით მოწესრიგების დაურღვევლად.
2.9. მოცემულია ზრდადობის მიხედვით დალაგებული ორი მასივი. შეადგინეთ პროგრამა, რომელიც ამ ორი მასივისაგან შექმნის მესამე მასივს, რომლის ელემენტებიც, ასევე, ზრდადობის მიხედვით იქნება დალაგებული.
2.10. შეადგინეთ პროგრამა, რომელიც მაგისტრების გვარებს ანბანის მიხედვით დაალაგებს.

კალენდარი

შეადგინეთ პროგრამა, რომელიც:
3.1. კვირის დღის ნომრის მიხედვით კვირის შესაბამისი დღის დასახელებას ქართულ ენაზე გამოიტანს ეკრანზე.
3.2. რიცხვის, თვისა და წლის მიხედვით კვირის დღეს განსაზღვრავს.
3.3. ეკრანზე ნაკიან წლებს გამოიტანს. ნაკიანია წელი თუ მისი ნომერი უნაშთოდ იყოფა 4-ზე.
3.4. განსაზღვრავს თუ რამდენჯერ მოხვდა თქვენი დაბადების დღე კვირის თითოეულ დღეზე.
3.5. განსაზღვრავს თვის იმ რიცხვებს, როდესაც მოუწევს ყოველი თვის უკანასკნელი კვირა დღე.
3.6. შეადგინეთ პროგრამა, რომელსაც თარიღი შემდეგი სახით მიეწოდება - 19.03.05. პროგრამამ ეკრანზე უნდა გამოიტანოს 2005 წლის 19 მარტი.
3.7. შეადგინეთ პროგრამა, რომელსაც თარიღი მიეწოდება. პროგრამამ უნდა განსაზღვროს შესაბამისი დღის ნომერი წლის დასაწყისიდან.
3.8. შეადგინეთ პროგრამა, რომელსაც ორი თარიღი მიეწოდება. პროგრამამ უნდა გამოთვალოს:
ა. დღეების რაოდენობა, რომელიც ამ ორ თარიღს შორის გავიდა;
ბ. თვეების რაოდენობა, რომელიც ამ ორ თარიღს შორის გავიდა;
გ. წლების რაოდენობა, რომელიც ამ ორ თარიღს შორის გავიდა;
3.9. შეადგინეთ პროგრამა, რომელსაც თარიღი მიეწოდება. პროგრამამ უნდა:
ა. შეამოწმოს თარიღის კორექტულობა. მაგალითად, არაკორექტულია 2005 წლის 30 თებერვალი;
ბ. განსაზღვროს თუ რამდენი დღე დარჩა წლის ბოლომდე;
გ. განსაზღვროს თუ რამდენი დღე გავიდა წლის დასაწყისიდან.
3.10. მოცემულია თვე და რიცხვი. შეადინეთ პროგრამა, რომელიც განსაზღვრავს თუ რომელ დღეზე მოდის ეს თარიღი თუ წელი არანაკიანია, 1 იანვარი კი - ხუთშაბათი.

პედაგოგიკა

შეადგინეთ პროგრამა, რომელიც:

4.1. შეამოწმებს გამრავლების ტაბულის ცოდნას.

4.2. შეამოწმებს საისტორიო თარიღების ცოდნას. პროგრამამ ეკრანზე უნდა გამოიტანოს ისტორიული მოვლენის დასახელება, თქვენ კი უნდა შეიტანოთ შესაბამისი თარიღი.

4.3. შეამოწმებს:

ა. გამრავლების ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში;

ბ. შეკრების ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში;

გ. გამოკლების ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში;

დ. გაყოფის ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში;

ე. ახარისხების ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში;

ვ. კვადრატული ფესვის ამოღების ოპერაციის შესრულების სისწორეს 100-ის ფარგლებში.

დანართი 2. სავარჯიშოების ამოხსნები

თავი 2. C++ ენის საფუძვლები

არითმეტიკული გამოსახულება

2.1.4.

```
{
   double x, y;

   x = Convert::ToDouble(textBox1->Text);
   y = ( 1 + x ) * ( x - Math::Sqrt(x - 1) / 4 ) / ( Math::Exp(x) + 1 / ( Math::Pow(x, 2) + 4 ) );
   label1->Text = Math::Round(y, 2).ToString();
}
```

2.1.10.

```
{
   int a, b, c, Perimetri;
   double Partobi;

   a = Convert::ToInt32(textBox1->Text);
   b = Convert::ToInt32(textBox2->Text);
   c = Convert::ToInt32(textBox3->Text);
   Perimetri = ( a + b + c ) / 2;
   Partobi = Math::Sqrt( Perimetri * ( Perimetri - a ) * ( Perimetri - b ) * ( Perimetri - c ) );
   label1->Text = Math::Round(Partobi, 2).ToString();
}
```

ლოგიკური გამოსახულება

2.2.3.

```
{
   bool y, x1, x2, x3, x4;

   x1 = Convert::ToBoolean(textBox1->Text);     // x1 = True
   x2 = Convert::ToBoolean(textBox2->Text);     // x2 = True
   x3 = Convert::ToBoolean(textBox3->Text);     // x3 = False
   x4 = Convert::ToBoolean(textBox4->Text);     // x4 = False
   y = !x1 && !x2 && x3 || x4;                  // y = False
   label1->Text = y.ToString();
}
```

თავი 3. მმართველი ოპერატორი

if ოპერატორი

3.1.3.

```
{
```

```
    int ricxvi;

    ricxvi = Convert::ToInt32(textBox1->Text);
    if ( ricxvi % 5 == 0 ) label1->Text = L"რიცხვი 5-ის ჯერადია";
      else label1->Text = L"რიცხვი 5-ის ჯერადი არ არის";
  }
```

3.1.8.

```
  {
    int ricxvi1, ricxvi2;

    ricxvi1 = Convert::ToInt32(textBox1->Text);
    ricxvi2 = Convert::ToInt32(textBox2->Text);
    if ( ricxvi1 >= ricxvi2 ) label1->Text = L"პირველი რიცხვი მეორეზე მეტია ან მისი ტოლია";
      else label1->Text = L"მეორე რიცხვი პირველზე მეტია";
  }
```

ჩადგმული if ოპერატორი

3.2.1.

```
  {
    int ricxvi1 = Convert::ToInt32(textBox1->Text);
    int ricxvi2 = Convert::ToInt32(textBox2->Text);
    int ricxvi3 = Convert::ToInt32(textBox3->Text);

    if ( ricxvi1 >= ricxvi2 )
          if ( ricxvi1 >= ricxvi3 ) label1->Text = L"პირველი რიცხვი მაქსიმალურია";
              else label1->Text = L"მესამე რიცხვი მაქსიმალურია";
       else
          if ( ricxvi2 >= ricxvi3 ) label1->Text = L"მეორე რიცხვი მაქსიმალურია";
              else label1->Text = L"მესამე რიცხვი მაქსიმალურია";
  }
```

3.2.8.

```
  {
    int ricxvi1 = Convert::ToInt32(textBox1->Text);
    int ricxvi2 = Convert::ToInt32(textBox2->Text);
    int ricxvi3 = Convert::ToInt32(textBox3->Text);

    if ( ricxvi1 < 0 ) label1->Text = L"სამ რიცხვს შორის არის უარყოფითი";
    else if ( ricxvi2 < 0 ) label1->Text = L"სამ რიცხვს შორის არის უარყოფითი";
      else if ( ricxvi3 < 0 ) label1->Text = L"სამ რიცხვს შორის არის უარყოფითი";
        else label1->Text = L"სამ რიცხვს შორის არ არის უარყოფითი";
  }
```

ეს ამოცანა შეიძლება გადაწყდეს ლოგიკის გამოყენებით:

```
  {
    int ricxvi1 = Convert::ToInt32(textBox1->Text);
    int ricxvi2 = Convert::ToInt32(textBox2->Text);
    int ricxvi3 = Convert::ToInt32(textBox3->Text);
```

```
    if ( ( ricxvi1 < 0 ) || ( ricxvi1 < 0 ) || ( ricxvi1 < 0 ) )
            label1->Text = L"სამ რიცხვს შორის არის უარყოფითი";
       else label1->Text = L"სამ რიცხვს შორის არ არის უარყოფითი";
  }
```

3.2.9.

```
  {
    int raodenoba = 0;
    int ricxvi1 = Convert::ToInt32(textBox1->Text);
    int ricxvi2 = Convert::ToInt32(textBox2->Text);
    int ricxvi3 = Convert::ToInt32(textBox3->Text);

    if ( ricxvi1 % 2 == 1 ) raodenoba++;
    if ( ricxvi2 % 2 == 1 ) raodenoba++;
    if ( ricxvi3 % 2 == 1 ) raodenoba++;
    label1->Text = raodenoba.ToString();
  }
```

ლოგიკა

3.3.2.

```
  {
    int ricxvi = Convert::ToInt32(textBox1->Text);
    if ( ( ricxvi >= 10 ) && ( ricxvi < 19 ) )
            label1->Text = L"რიცხვი მოთავსებულია მითითებულ დიაპაზონში";
     else label1->Text = L"რიცხვი მოთავსებული არ არის მითითებულ დიაპაზონში";
  }
```

3.3.3.

```
  {
    int ricxvi = Convert::ToInt32(textBox1->Text);
    if ( ( ricxvi < 25 ) || ( ricxvi > 100 ) )
            label1->Text = L"რიცხვი მოთავსებულია მითითებულ დიაპაზონში";
       else label1->Text = L"რიცხვი მოთავსებული არ არის მითითებულ დიაპაზონში";
  }
```

for, while, do-while ოპერატორები

3.4.1.

```
  {
    double jami = 0;
    int mnishvneli, minusi = -1;

    for (mnishvneli = 1; mnishvneli <= 5; mnishvneli++)
    {
       minusi *= -1;
       jami += minusi * ( 1.0  / mnishvneli);
    }
    label1->Text = Math::Round(jami, 2).ToString();
  }
```

3.4.13.

```
{
   double ricxvi, jami = 0;
   int mnishvneli;

   ricxvi = Convert::ToDouble(textBox1->Text);
   for ( mnishvneli = 1; mnishvneli <= 7; mnishvneli += 2 )
      jami += Math::Pow(ricxvi, mnishvneli) / mnishvneli;
   jami *= 2;
   label1->Text = Math::Round(jami, 2).ToString();
}
```

3.4.16.

```
{
   int jami = 0;
   for ( int ricxvi = 10; ricxvi <= 20; ricxvi++ )
      if ( ricxvi % 2 == 1 ) jami += ricxvi;
   label1->Text = jami.ToString();
}
```

ამ ამოცანის გადაწყვეტის მეორე გზა

```
{
   int jami = 0;
   for ( int ricxvi = 11; ricxvi <= 20; ricxvi += 2 )
      jami += ricxvi;
   label1->Text = jami.ToString();
}
```

continue, break ოპერატორები

3.5.1.

```
{
   int jami = 0;

   for ( int ricxvi = 1; ricxvi <= 20; ricxvi++ )
      if ( jami > 24 ) break;
         else jami += ricxvi;
   label1->Text = jami.ToString();
}
```

3.5.2.

```
{
   int raodenoba = 0;

   for ( int ricxvi = 6; ricxvi <= 79; ricxvi++ )
   {
      if ( ricxvi % 4 == 0 ) raodenoba++;
      if ( ricxvi > 50 ) break;
```

```
    }
    label1->Text = raodenoba.ToString();
}
```

თავი 4. მასივი, სტრიქონი და ბიტობრივი ოპერაცია

ერთგანზომილებიანი მასივი

4.1.5.

```
{
    array<int>^ masivi = gcnew array<int> { 1, -5, 20, -2, 4};
    int max = masivi[0], maxind = 0;

    for ( int indexi = 1; indexi < masivi.Length; indexi++ )
        if ( max < masivi[indexi] )
        {
            max = masivi[indexi];
            maxind = indexi;
        }
    label1->Text = L"მასივის მაქსიმალური ელემენტია - " + max.ToString() + "\n" +
                   L"მაქსიმალური ელემენტის ინდექსია - " + maxind.ToString();
    for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        label2->Text += masivi[indexi].ToString() + "   ";
}
```

4.1.8.

```
{
    array<int>^ masivi = gcnew array<int> { 1, -5, 20, -2, 4};
    int jami = 0;

    for ( int indexi = 0; indexi < masivi.Length; indexi++ )
        if ( masivi[indexi] >= 0 ) jami += masivi[indexi];
    label1->Text = L"დადებითი ელემენტების ჯამია - " + jami.ToString();
    for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        label2->Text += masivi[indexi].ToString() + "   ";
}
```

4.1.9.

```
{
    array<int>^ masivi = gcnew array<int> { 1, -5, 20, -2, 4};
    int raodenoba = 0;

    for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        if ( masivi[indexi] < 0 ) raodenoba++;
    label1->Text = L"უარყოფითი ელემენტების რაოდენობაა - " + raodenoba.ToString();
    for ( int indexi = 0; indexi < masivi->Length; indexi++ )
        label2->Text += masivi[indexi].ToString() + "   ";
}
```

4.1.12.

```
{
   array<int>^ masivi = gcnew array<int> { 1, -5, 20, -2, 4};
   int jami = 0;

   for ( int indexi = 0; indexi < masivi->Length; indexi += 2 )
      if ( masivi[indexi] >= 0 ) jami += masivi[indexi];
   label1->Text = L"ლუწი ინდექსის მქონე ელემენტების ჯამია - " + jami.ToString();
   for ( int indexi = 0; indexi < masivi->Length; indexi++ )
      label2->Text += masivi[indexi].ToString() + "   ";
}
```

4.1.15.

```
{
   array<int>^ masivi = gcnew array<int> { 1 , 5 , 20 , -2 , 4 , 75 , -3 , 85 , 9 , 21 };
   int indexi;

   for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      if ( masivi[indexi] < 0 ) break;
         else continue;
   label1->Text = L"პირველი უარყოფითი ელემენტია -  " +
            masivi[indexi].ToString() + L"   მისი ინდექსია - " + indexi.ToString();
}
```

4.1.17.

```
{
   array<int>^ masivi = gcnew array<int> { 1 , 5 , -20 , -2 , 4 , 75 , -3 , -85 , 9 , -21 };
   int indexi, max = masivi[0], maxind = 0;

   for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      if ( max < masivi[indexi] ) max = masivi[indexi];
   max *= -1;
   for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      if ( ( masivi[indexi] < 0 ) && ( max < masivi[indexi] ) )
      {
         max = masivi[indexi];
         maxind = indexi;
      }
   label1->Text = L"მაქსიმალური უარყოფითი რიცხვია -   " +
            max.ToString() + L"\nმისი ინდექსია - " + maxind.ToString();
   for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label2->Text += masivi[indexi].ToString() + "   ";
}
```

4.1.22.

```
{
   array<int>^ masivi = gcnew array<int> { 1, 5, -20, -2, 4, 75, -3, -85, 9, -21 };
   int indexi, min, max, raodenoba = 0;

   min = Convert::ToInt32(textBox1->Text);
```

```
        max = Convert::ToInt32(textBox2->Text);
        for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
          if ( ( min <= masivi[indexi] ) && ( max >= masivi[indexi] ) ) raodenoba++;
        label1->Text = raodenoba.ToString();
        for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
          label2->Text += masivi[indexi].ToString() + "   ";
    }
```

4.1.25.

```
{
array<int>^ masivi = gcnew array<int> { 3, 9, 6, -1, 12, 10, 17, 4, 6, 19 };
int ind, jami = 0;
//        საწყისი ელემენტის ინდექსის შეტანა
int min_ind = Convert::ToInt32(textBox1->Text);
//        საბოლოო ელემენტის ინდექსის შეტანა
int max_ind = Convert::ToInt32(textBox2->Text);

for (ind = min_ind; ind <= max_ind; ind++)
    jami += masivi [ind];
label3->Text = jami.ToString();
}
```

4.1.26.

```
{
array<int>^ masivi = gcnew array<int> { 3, 9, 6, -1, 12, 10, 17, 4, 6, 19 };
int ind, max;
//        პირველი ელემენტის ინდექსი
int min_ind = Convert::ToInt32(textBox1->Text);
//        მეორე ელემენტის ინდექსი
int max_ind = Convert::ToInt32(textBox2->Text);
max = masivi[min_ind];
for (ind = min_ind; ind <= max_ind; ind++)
    if ( max < masivi[ind] )  max = masivi[ind];
label3->Text = max.ToString();
}
```

4.1.28.

```
{
label1->Text = "";
array<int>^ masivi1 = gcnew array<int> { 3, 29, 6, -1, 12, 40, 17, 64, 6, 79 };
array<int>^ masivi2 = gcnew array<int> (masivi1.Length);
int ind1, ind2 = 0, ricxvi;
ricxvi = Convert::ToInt32(textBox1->Text);
for ( ind1 = 0; ind1 < masivi1->Length; ind1++ )
    if ( masivi1[ind1] < ricxvi ) masivi2[ind2++] = masivi1[ind1];
for ( ind1 = 0; ind1 < ind2; ind1++ )
    label1->Text += masivi2[ind1].ToString() + "   ";
}
```

4.1.31.

```
    {
```

```
    array<int>^ masivi1 = gcnew array<int> { 1 , -5 , 20 , -2 , 4, 75, -3, 85, 9, 21 };
   array<int>^ masivi2 = gcnew array<int> (masivi1.Length);
    int indexi1 , indexi2 = 0;

    for ( indexi1 = 0 ; indexi1 < masivi1->Length ; indexi1++ )
      if ( masivi1[indexi1] % 5 == 0 ) masivi2[indexi2++] = masivi1[indexi1];

    for ( indexi1 = 0 ; indexi1 < masivi1->Length ; indexi1++ )
      label1->Text += masivi1[indexi1].ToString() + "   ";
    for ( indexi1 = 0 ; indexi1 < indexi2 ; indexi1++ )
      label2->Text += masivi2[indexi1].ToString() + "   ";
  }
```

4.1.36.

```
  {
    array<int>^ masivi1 = gcnew array<int> { 1 , -5 , 20 , -2 , 4 , 75 , -3 , 85 , 9 , 21 };
    array<int>^ masivi2 = gcnew array<int> (masivi1.Length);
    int indexi1 , indexi2 = 0;

    for ( indexi1 = 0 ; indexi1 < masivi1->Length ; indexi1++ )
      if ( masivi1[indexi1] >= 0 ) masivi2[indexi2++] = indexi1;

    for ( indexi1 = 0 ; indexi1 < masivi1->Length ; indexi1++ )
      label1->Text += masivi1[indexi1].ToString() + "   ";
    for ( indexi1 = 0 ; indexi1 < indexi2 ; indexi1++ )
      label2->Text += masivi2[indexi1].ToString() + "   ";
  }
```

4.1.45.

```
  {
    array<int>^ masivi = gcnew array<int> { 1 , -5 , 20 , -2 , 4 , 75 , -3 , 85 , 9 , 21 };
    int indexi, temp;

    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label1->Text += masivi[indexi].ToString() + "   ";
    for ( indexi = 0 ; indexi < masivi->Length / 2 ; indexi++ )
    {
      temp = masivi[indexi];
      masivi[indexi] = masivi[masivi->Length - 1 - indexi];
      masivi[masivi->Length - 1 - indexi] = temp;
    }
    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label2->Text += masivi[indexi].ToString() + "   ";
  }
```

4.1.48.

```
  {
    array<int>^ masivi = gcnew array<int> { 1, -5, 0, -2, 0, 0, -3, 0, 9, 0 };
    int indexi, temp, Alami;
```

```
    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label1->Text += masivi[indexi].ToString() + "   ";
  M1: Alami = 0;
    for ( indexi = 0 ; indexi < masivi->Length - 1; indexi++ )
      if ( ( masivi[indexi] == 0 ) && ( masivi[indexi + 1] != 0 ) )
    {
      temp = masivi[indexi];
      masivi[indexi] = masivi[indexi + 1];
      masivi[indexi + 1] = temp;
      Alami = 1;
    }
    if ( Alami == 1 ) goto M1;
    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label2->Text += masivi[indexi].ToString() + "   ";
  }
```

4.1.49.

```
  {
    array<int>^ masivi = gcnew array<int> { 1, 5, 2, 4, 7 };
    int indexi, temp, Zvris_Raodenoba = Convert::ToInt32(textBox1->Text);

    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
        label1->Text += masivi[indexi].ToString() + "   ";
    for ( int i = 1 ; i <= Zvris_Raodenoba ; i++ )
    {
      temp = masivi[masivi->Length - 1];
      for ( indexi = 0 ; indexi < masivi->Length - 1 ; indexi++ )
        masivi[masivi->Length - 1 - indexi] = masivi[masivi->Length - 2 - indexi];
      masivi[0] = temp;
    }
    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label2->Text += masivi[indexi].ToString() + "   ";
  }
```

4.1.50.

```
  {
    array<int>^ masivi = gcnew array<int> { 1, 5, 2, 4, 7 };
    int indexi, temp, Zvris_Raodenoba = Convert::ToInt32(textBox1->Text);

    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
      label1->Text += masivi[indexi].ToString() + "   ";
    for ( int i = 1 ; i <= Zvris_Raodenoba ; i++ )
    {
      temp = masivi[0];
      for ( indexi = 0 ; indexi < masivi->Length - 1 ; indexi++ )
        masivi[indexi] = masivi[indexi + 1];
      masivi[masivi->Length - 1] = temp;
    }
    for ( indexi = 0 ; indexi < masivi->Length ; indexi++ )
```

```
        label2->Text += masivi[indexi].ToString() + "   ";
    }
```

ორგანზომილებიანი მასივი

4.2.2.

```
    {
        array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
        int striqoni, sveti, minstriqoni = 0, minsveti = 0, min = masivi[0, 0];

        for ( striqoni = 0; striqoni < 3; striqoni++ )
            for ( sveti = 0; sveti < 3; sveti++ )
                if ( min > masivi[striqoni, sveti] )
                {
                    min = masivi[striqoni, sveti];
                    minstriqoni = striqoni;
                    minsveti = sveti;
                }
         label1->Text = L"მინიმალური ელემენტია - " + min.ToString() +
                    L"\nსტრიქონის ინდექსია - " + minstriqoni.ToString() +
                    L"\nსვეტის ინდექსია - " + minsveti.ToString();
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        {
            for ( sveti = 0 ; sveti < 3 ; sveti++ )
                label2->Text += masivi[striqoni, sveti].ToString() + "  ";
            label2->Text += "\n";
        }
    }
```

4.2.7.

```
    {
        array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
        int striqoni, sveti, jami = 0;

        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
            for ( sveti = 0 ; sveti < 3 ; sveti++ )
                if ( masivi[striqoni, sveti] % 2 == 1 )
                    jami += masivi[striqoni, sveti];
        label1->Text = jami.ToString();
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        {
            for ( sveti = 0 ; sveti < 3 ; sveti++ )
                label2->Text += masivi[striqoni, sveti].ToString() + "  ";
            label2->Text += "\n";
        }
    }
```

4.2.8.

```
    {
        array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
```

```
        int striqoni, sveti, raodenoba = 0;

        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
           for ( sveti = 0 ; sveti < 3 ; sveti++ )
              if ( masivi[striqoni, sveti] % 2 == 0 )  raodenoba++;
        label1->Text = raodenoba.ToString();
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        {
           for ( sveti = 0 ; sveti < 3 ; sveti++ )
              label2->Text += masivi[striqoni, sveti].ToString() + "  ";
           label2->Text += "\n";
        }
     }
```

4.2.11.

```
     {
        array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
        int striqoni, sveti, jami = 0;

        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
                 jami += masivi[striqoni, striqoni];
        label1->Text = jami.ToString();
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        {
           for ( sveti = 0 ; sveti < 3 ; sveti++ )
              label2->Text += masivi[striqoni, sveti].ToString() + "  ";
           label2->Text += "\n";
        }
     }
```

4.2.12.

```
        {
        array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
        int striqoni, sveti, namravli = 1;

        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
           namravli *= masivi[striqoni, 2 - striqoni];
        label1->Text = namravli.ToString();
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        {
           for ( sveti = 0 ; sveti < 3 ; sveti++ )
              label2->Text += masivi[striqoni, sveti].ToString() + "  ";
           label2->Text += "\n";
        }
      }
```

4.2.20.

```
     {
        array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
        array<int>^ masivi2 = gcnew array<int> (masivi1.Length);
```

```
    int indexi2 = 0, striqoni, sveti;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
       for ( sveti = 0 ; sveti < 3 ; sveti++ )
          if ( masivi1[striqoni, sveti] % 2 == 0 ) masivi2[indexi2++] = masivi1[striqoni, sveti];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
       for ( sveti = 0 ; sveti < 3 ; sveti++ )
          label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
       label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < indexi2 ; striqoni++ )
       label2->Text += masivi2[striqoni].ToString() + "   ";
 }
```

4.2.22.

```
 {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
    array<int>^ masivi2 = gcnew array<int> (masivi1->Length);
    int striqoni, sveti;

    for ( sveti = 0 ; sveti < 3 ; sveti++ )
       for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
          masivi2[sveti] += masivi1[striqoni, sveti];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
       for ( sveti = 0 ; sveti < 3 ; sveti++ )
          label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
       label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
       label2->Text += masivi2[striqoni].ToString() + "   ";
 }
```

4.2.23.

```
 {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
    array<int>^ masivi2 = gcnew array<int> { 1, 1, 1 };
    int striqoni, sveti;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
       for ( sveti = 0 ; sveti < 3 ; sveti++ )
          masivi2[striqoni] *= masivi1[striqoni, sveti];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
       for ( sveti = 0 ; sveti < 3 ; sveti++ )
          label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
       label1->Text += "\n";
    }
```

```
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        label2->Text += masivi2[striqoni].ToString() + "   ";
}
```

4.2.25.

```
{
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 1, -2, 3 }, { 4, 5, -6 }, { 7, 8, 9 } };
    array<int>^ masivi2 = gcnew array<int>(3);
    int striqoni, sveti;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        for ( sveti = 0 ; sveti < 3 ; sveti++ )
            if ( masivi1[striqoni, sveti] %2 == 1 ) masivi2[striqoni] += masivi1[striqoni, sveti];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
        for ( sveti = 0 ; sveti < 3 ; sveti++ )
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
        label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        label2->Text += masivi2[striqoni].ToString() + "   ";
}
```

4.2.26.

```
{
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { -1, 4, 13 }, { 7, 5, -6 }, { 4, 8, 9 } };
    array<int>^ masivi2 = gcnew array<int>(3);
    int striqoni, sveti, max;

    for ( sveti = 0 ; sveti < 3 ; sveti++ )
    {
        max = masivi1[0, sveti];
        for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
            if ( max < masivi1[striqoni, sveti] ) max = masivi1[striqoni, sveti];
        masivi2[sveti] = max;
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
        for ( sveti = 0 ; sveti < 3 ; sveti++ )
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
        label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        label2->Text += masivi2[striqoni].ToString() + "   ";
}
```

4.2.28.

```
{
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 11, -2, 3 }, { 4, 15, -6 }, { -7, 8, 9 } };
    array<int>^ masivi2 = gcnew array<int>(3);
```

```
    int striqoni, sveti, max = masivi1[0,0], maxstriqoni = 0;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      if ( max < masivi1[striqoni, striqoni] )
      {
        max = masivi1[striqoni, striqoni];
        maxstriqoni = striqoni;
      }
    for ( sveti = 0 ; sveti < 3 ; sveti++ )
      masivi2[sveti] = masivi1[maxstriqoni, sveti];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
      label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      label2->Text += masivi2[striqoni].ToString() + "   ";
  }
```

4.2.34.

```
  {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    array<int>^ masivi2 = gcnew array<int>(3);
    int striqoni, sveti;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      masivi2[striqoni] = masivi1[striqoni, striqoni];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
      label1->Text += "\n";
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      label2->Text += masivi2[striqoni].ToString() + "  ";
  }
```

4.2.35.

```
  {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    array<int>^ masivi2 = gcnew array<int>(3);
    int striqoni, sveti;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      masivi2[striqoni] = masivi1[striqoni, 2 - striqoni];
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
```

```
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
         label1->Text += "\n";
      }
      for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
         label2->Text += masivi2[striqoni].ToString() + "  ";
   }
```

4.2.36.

```
   {
      array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
      array<double>^ masivi2 = gcnew array<double>(3);
      int striqoni, sveti, indexi2 = 0;
      double sashualo;

      for ( sveti = 0; sveti < 3; sveti += 2 )
         {
            sashualo = 0;
            for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
            sashualo += masivi1[striqoni, sveti];
            masivi2[indexi2++] = sashualo / 3;
         }
      for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      {
         for ( sveti = 0 ; sveti < 3 ; sveti++ )
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
         label1->Text += "\n";
      }
      for ( striqoni = 0 ; striqoni < indexi2 ; striqoni++ )
         label2->Text += masivi2[striqoni].ToString() + "  ";
   }
```

4.2.37.

```
   {
      array<int,2>^ masivi1 = gcnew array<int,2> (4, 4)
                                                { { 10, 21, 3, 2 }, { 4, 51, 6, 5 }, { 7, 3, 9, 6 }, { 4, 1, 8, 9 } };
      array<double>^ masivi2 = gcnew array<double>(4);
      int striqoni, sveti, indexi2 = 0;
      double sashualo;

      for ( striqoni = 1 ; striqoni < 4 ; striqoni += 2 )
      {
         sashualo = 1;
         for ( sveti = 0 ; sveti < 4 ; sveti++ )
            sashualo *= masivi1[striqoni, sveti];
         masivi2[indexi2++] = sashualo / 4;
      }
      for ( striqoni = 0 ; striqoni < 4 ; striqoni++ )
      {
         for ( sveti = 0 ; sveti < 4 ; sveti++ )
```

```
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
         label1->Text += "\n";
      }
      for ( striqoni = 0 ; striqoni < indexi2 ; striqoni++ )
         label2->Text += masivi2[striqoni].ToString() + "  ";
   }
```

4.2.38.

```
   {
      array<int,2>^ masivi1 = gcnew array<int,2> (4, 4)
                                                { { 10, 21, 3, 2 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 4, 1, 8, 9 } };
      array<int>^ masivi2 = gcnew array<int>(4);
      int striqoni, sveti, indexi2 = 0, raodenoba = 0;

      for ( striqoni = 0 ; striqoni < 4 ; striqoni++ )
      {
         raodenoba = 0;
         for ( sveti = 0 ; sveti < 4 ; sveti++ )
            if ( masivi1[striqoni, sveti] == 0 ) raodenoba++;
         if ( raodenoba == 4 ) masivi2[indexi2++] = striqoni;
      }
      for ( striqoni = 0 ; striqoni < 4 ; striqoni++ )
      {
         for ( sveti = 0 ; sveti < 4 ; sveti++ )
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
         label1->Text += "\n";
      }
      for ( striqoni = 0 ; striqoni < indexi2 ; striqoni++ )
         label2->Text += masivi2[striqoni].ToString() + "  ";
   }
```

4.2.39.

```
   {
      array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 61 }, { 7, 3, 9 } };
      array<int>^ masivi2 = gcnew array<int>(3);
      int striqoni, sveti, raodenoba = 0, indexi2 = 0;

      for ( sveti = 0 ; sveti < 3 ; sveti++ )
      {
         raodenoba = 0;
         for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
            if ( masivi1[striqoni, sveti] % 2 == 1 ) raodenoba++;
         if ( raodenoba == 3 ) masivi2[indexi2++] = sveti;
      }
      for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      {
         for ( sveti = 0 ; sveti < 3 ; sveti++ )
            label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
         label1->Text += "\n";
```

```
    }
    for ( striqoni = 0 ; striqoni < indexi2 ; striqoni++ )
      label2->Text += masivi2[striqoni].ToString() + "  ";
  }
```

4.2.46.

```
  {
    label1->Text = ""; label2->Text = "";
    array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    int striqoni, sveti, temp;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label1->Text += masivi[striqoni, sveti].ToString() + "  ";
      label1->Text += "\n";
    }
     //        შეგვაქვს სვეტის ინდექსი, რომლის მნიშვნელობა უნდა იყოს 3-ზე ნაკლები
     sveti = Convert::ToInt32(textBox1->Text);
     if ( sveti >= 3 )
     {
              label1->Text = L"არასწორი ინდექსი, გაიმეორეთ შეტანა";
              return;
     }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      temp = masivi[striqoni, sveti];
      masivi[striqoni, sveti] = masivi[striqoni, 2];
      masivi[striqoni, 2] = temp;
    }
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label2->Text += masivi[striqoni, sveti].ToString() + "  ";
      label2->Text += "\n";
    }
  }
```

4.2.47.

```
  {
    array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    int striqoni, sveti, temp;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label1->Text += masivi[striqoni, sveti].ToString() + "  ";
      label1->Text += "\n";
    }
```

```
//          შეგვაქვს სტრიქონის ინდექსი, რომლის მნიშვნელობა უნდა იყოს 3-ზე ნაკლები
striqoni = Convert::ToInt32(textBox1->Text);
if ( striqoni >= 3 )
        {
                label1->Text = L"არასწორი ინდექსი, გაიმეორეთ შეტანა";
                return;
        }
for ( sveti = 0 ; sveti < 3 ; sveti++ )
{
   temp = masivi[striqoni, sveti];
   masivi[striqoni, sveti] = masivi[0, sveti];
   masivi[0, sveti] = temp;
}
for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
{
   for ( sveti = 0 ; sveti < 3 ; sveti++ )
      label2->Text += masivi[striqoni, sveti].ToString() + "  ";
   label2->Text += "\n";
}
}
```

4.2.65.

```
{
array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 11, -2, 3 }, { 4, -15, -6 }, { -7, 8, 9 } };
array<int>^ masivi2 = gcnew array<int> (3);
int striqoni, sveti, min, minind, raodenoba = 0;

for ( sveti = 0 ; sveti < 3 ; sveti++ )
{
   raodenoba = 0;
   for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      if ( masivi1[striqoni, sveti] >= 0) raodenoba++;
   masivi2[sveti] = raodenoba;
}
min = masivi2[0];
minind = 0;
for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
   if ( min > masivi2[striqoni] )
   {
      min = masivi2[striqoni];
      minind = striqoni;
   }
for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
{
   for ( sveti = 0 ; sveti < 3 ; sveti++ )
      label1->Text += masivi1[striqoni, sveti].ToString() + "  ";
   label1->Text += "\n";
}
```

```
    label2->Text = minind.ToString();
  }
```

4.2.73.

```
  {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    array<int>^ masivi2 = gcnew array<int> (3);
    int striqoni, sveti, max, maxind;

    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      for ( sveti = 0; sveti < 3; sveti++ )
      masivi2[striqoni] += masivi1[striqoni, sveti];
    max = masivi2[0];
    maxind = 0;
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      if ( max < masivi2[striqoni] )
      {
        max = masivi2[striqoni];
        maxind = striqoni;
      }
      label1->Text = maxind.ToString();
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
      for ( sveti = 0 ; sveti < 3 ; sveti++ )
        label2->Text += masivi1[striqoni, sveti].ToString() + "  ";
      label2->Text += "\n";
    }
  }
```

4.2.74.

```
  {
    array<int,2>^ masivi1 = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
    array<int>^ masivi2 = gcnew array<int> (3);
    int striqoni, sveti, max, maxind;

    for ( sveti = 0 ; sveti < 3 ; sveti++ )
      for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
        masivi2[sveti] += masivi1[striqoni, sveti];
    max = masivi2[0];
    maxind = 0;
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
      if ( max < masivi2[striqoni] )
      {
        max = masivi2[striqoni];
        maxind = striqoni;
      }
    label1->Text = maxind.ToString();
    for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
    {
```

```
            for ( sveti = 0 ; sveti < 3 ; sveti++ )
               label2->Text += masivi1[striqoni, sveti].ToString() + "  ";
            label2->Text += "\n";
         }
      }
```

4.2.78.

```
      {
         array<int,2>^ masivi = gcnew array<int,2> (3, 3) { { 10, 21, 3 }, { 4, 51, 6 }, { 7, 3, 9 } };
         int striqoni, sveti, temp;
          //          ინდექსების მნიშვნელობები უნდა იყოს 3-ზე ნაკლები
         striqoni = Convert::ToInt32(textBox1->Text);
         sveti = Convert::ToInt32(textBox2->Text);
         temp = masivi[striqoni, sveti];
         for ( int striqoni1 = 0; striqoni1 < 3; striqoni1++ )
            masivi[striqoni1, sveti] = 0;
         for ( int sveti1 = 0 ; sveti1 < 3 ; sveti1++ )
            masivi[striqoni, sveti1] = 0;
         masivi[striqoni, sveti] = temp;
         for ( striqoni = 0 ; striqoni < 3 ; striqoni++ )
         {
            for ( sveti = 0 ; sveti < 3 ; sveti++ )
               label1->Text += masivi[striqoni, sveti].ToString() + "  ";
            label1->Text += "\n";
         }
      }
```

თავი 6. კლასი, ინკაფსულაცია, მეთოდი

კლასი. ინკაფსულაცია

6.1.1.

```
ref class Tvitmfrinavi
{
int banis_tevadoba;
int manzili;
public : int mgzavrebis_raodenoba;
int gayiduli_bilitebi;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
Tvitmfrinavi^ obieqti_1 = gcnew Tvitmfrinavi();
obieqti_1->mgzavrebis_raodenoba = Convert::ToInt32(textBox1->Text);
obieqti_1->gayiduli_bilitebi = Convert::ToInt32(textBox2->Text);
label1->Text = obieqti_1->mgzavrebis_raodenoba.ToString();
label2->Text = obieqti_1->gayiduli_bilitebi.ToString();
}
```

6.1.2.

```
ref class Studenti
{
System::String^ gvari;
System::String^ saxeli;
int asaki;
public : System::String^ universitetis_dasaxeleba;
          int kursi;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
Studenti^ obieqti_1 = gcnew Studenti();
obieqti_1->universitetis_dasaxeleba = textBox1->Text;
obieqti_1->kursi = Convert::ToInt32(textBox2->Text);
label1->Text = obieqti_1->universitetis_dasaxeleba;
label2->Text = obieqti_1->kursi.ToString();
}
```

ფუნქცია

6.2.1.

```
ref class Studenti_1
      {
         public : double Sashualo_qula(array<int>^ masivi1)
         {
            int jami = 0;
            for (int ind = 0; ind < masivi1->Length; ind++)
               jami += masivi1[ind];
            return jami / masivi1->Length;
         }
      };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
array<int>^ masivi = gcnew array<int> { 60, 87, 71, 90, 94, 58, 83, 57, 70, 65 };
double shedegi;
Studenti_1^ obieqti_1 = gcnew Studenti_1();
shedegi = obieqti_1->Sashualo_qula(masivi);
label1->Text = shedegi.ToString();
}
```

6.2.2.

```
ref class Studenti_2
      {
         System::String^ gvari;
         System::String^ saxeli;
         int asaki;
         void minicheba(System::String^ par1, System::String^ par2, int par3)
```

```
        {
            gvari = par1;
            saxeli = par2;
            asaki = par3;
        }
         public : void gadacema(System::String^ par1, System::String^ par2, int par3)
        {
             minicheba(par1, par2, par3);
        }
         void gamotana(System::Windows::Forms::Label^ lab)
        {
            lab->Text = L"სტუდენტის გვარი: " + gvari + L"\nსტუდენტის სახელი: " + saxeli +
                        L"\nსტუდენტის ასაკი: " + asaki.ToString();
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
         Studenti_2^ obieqti_1 = gcnew Studenti_2();
        System::String^ gvari = textBox1->Text;
        System::String^ saxeli = textBox2->Text;
        int asaki = Convert::ToInt32(textBox3->Text);
        obieqti_1->gadacema(gvari, saxeli, asaki);
        obieqti_1->gamotana(label1);
    }
```

6.2.3.

```
ref class Matarebeli
    {
        int vagonebis_raodenoba;
        int mgzavrebis_raodenoba;
        public : double biletis_fasi;
        int gayiduli_biletebis_raodenoba;
         void minicheba(double par1, int par2, int par3, int par4)
        {
            biletis_fasi = par1;
            gayiduli_biletebis_raodenoba = par2;
            vagonebis_raodenoba = par3;
            mgzavrebis_raodenoba = par4;
        }
        void gamotana()
        {
         System::Windows::Forms::MessageBox::Show(L"ბილეთის ფასი - " + biletis_fasi.ToString() +
            L"\nგაყიდული ბილეთების რაოდენობა - " + gayiduli_biletebis_raodenoba.ToString() +
            L"\nვაგონების რაოდენობა - " + vagonebis_raodenoba.ToString() +
            L"\nმგზავრების რაოდენობა - " + mgzavrebis_raodenoba.ToString());
        }
        double mogeba()
```

```
        {
            return biletis_fasi * gayiduli_biletebis_raodenoba;
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
        int vagonebis_raodenoba = Convert::ToInt32(textBox3->Text);
        int mgzavrebis_raodenoba = Convert::ToInt32(textBox4->Text);
        double shedegi;
        Matarebeli^ obieqti_1 = gcnew Matarebeli();
        obieqti_1->biletis_fasi = Convert::ToDouble(textBox1->Text);
        obieqti_1->gayiduli_biletebis_raodenoba = Convert::ToInt32(textBox2->Text);
        obieqti_1->minicheba(obieqti_1->biletis_fasi, obieqti_1->gayiduli_biletebis_raodenoba,
                             vagonebis_raodenoba, mgzavrebis_raodenoba);
        obieqti_1->gamotana();
        shedegi = obieqti_1->mogeba();
        label1->Text = L"ბილეთების გაყიდვით მიღებულ თანხა = " + shedegi.ToString();
    }
```

კონსტრუქტორი

6.3.1.

```
ref class Tvitmfrinavi
    {
        int bakis_tevadoba;
        int manzili_1_litri;
        public : Tvitmfrinavi(int par1, int par2)
        {
            bakis_tevadoba = par1;
            manzili_1_litri = par2;
        }
        void Gamotana(System::Windows::Forms::Label^ lab)
        {
            lab->Text = L"ავზის ტევადობა = " + bakis_tevadoba.ToString() +
                        L"\n1 ლიტრი საწვავით გავლილი მანძილი = " + manzili_1_litri.ToString();
        }
       int Gamotvla()
        {
            return bakis_tevadoba * manzili_1_litri;
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
        int bakis_tevadoba = Convert::ToInt32(textBox1->Text);
        int manzili_1_litri = Convert::ToInt32(textBox2->Text);
        Tvitmfrinavi^ obieqti_1 = gcnew Tvitmfrinavi(bakis_tevadoba, manzili_1_litri);
```

```
        obieqti_1->Gamotana(label1);
        int shedegi = obieqti_1->Gamotvla();
        label2->Text = L"სავსე ავზით გავლილი მანძილი = " + shedegi.ToString();
    }
```

6.3.3.

```
ref class Martkutxedi
    {
        int perimetri;
        int fartobi;
        public : int gverdi_1;
        int gverdi_2;
        Martkutxedi(int par1, int par2)
        {
          gverdi_1 = par1;
          gverdi_2 = par2;
          perimetri = ( gverdi_1 + gverdi_2 ) * 2;
          fartobi = gverdi_1 * gverdi_2;
        }
        void Gamotana(System::Windows::Forms::Label^ lab1)
        {
          lab1->Text = L"პერიმეტრი = " + perimetri.ToString() + L"\nფართობი = " + fartobi.ToString();
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
        int gverdi_1 = Convert::ToInt32(textBox1->Text);
        int gverdi_2 = Convert::ToInt32(textBox2->Text);
        Martkutxedi^ obieqti_1 = gcnew Martkutxedi(gverdi_1, gverdi_2);
        obieqti_1->Gamotana(label1);
    }
```

this მიმთითებელი

6.4.1.

```
ref class Chemi_Klasi
    {
       int ricxvi1, ricxvi2, ricxvi3;
        public : int Gamotvla(int ricxvi1, int ricxvi2, int ricxvi3)
        {
          this->ricxvi1 = ricxvi1;
          this->ricxvi2 = ricxvi2;
          this->ricxvi3 = ricxvi3;
          return this->ricxvi1 + this->ricxvi2 + this->ricxvi3;
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
```

```
    {
        Chemi_Klasi^ obieqti_1 = gcnew Chemi_Klasi();
        int cvladi1 = Convert::ToInt32(textBox1->Text);
        int cvladi2 = Convert::ToInt32(textBox2->Text);
        int cvladi3 = Convert::ToInt32(textBox3->Text);

        int shedegi = obieqti_1->Gamotvla(cvladi1, cvladi2, cvladi3);
        label1->Text = shedegi.ToString();
    }
```

6.4.3.

```
ref class Chemi_Klasi1
    {
       array<int>^ masivi2;
       public : int Gamotvla(array<int>^ masivi2)
       {
          int ind;
          this->masivi2 = masivi2;
          for (ind = 0 ; ind < masivi2->Length ; ind++)
              if (masivi2[ind] < 0) break;
           return this->masivi2[ind];
       }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
       Chemi_Klasi1^ obieqti_1 = gcnew Chemi_Klasi1();
       array<int>^ masivi1 = gcnew array<int> { 5, 2, -3, 6, -1, 9, -8 };

       int shedegi = obieqti_1->Gamotvla(masivi1);
       label1->Text = shedegi.ToString();
    }
```

static მოდიფიკატორი

6.5.1.

```
ref class Otxkutxedi  {
        static int statikuri_simagle;
        static int statikuri_sigane;
        int perimetri;
                int chveulebrivi_Perimetri() {
                perimetri = ( statikuri_simagle + statikuri_sigane ) * 2;
                return perimetri;
        }
public :
          Otxkutxedi(int par1, int par2) {
                statikuri_simagle = par1;
                statikuri_sigane = par2;
}
```

```
        static int statikuri_Funqcia(Otxkutxedi^ obj) {
                return obj->chveulebrivi_Perimetri();
        }
};
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Otxkutxedi^ obj1 = gcnew Otxkutxedi(ricxvi1, ricxvi2);
int shedegi = Otxkutxedi::statikuri_Funqcia(obj1);
label3->Text = shedegi.ToString();
}
```

6.5.2.

```
class ChemiKlasi_2
    {
        public static int[] masivi_1;
        public ChemiKlasi_2(int[] masivi_2)
        {
            masivi_1 = masivi_2;
        }
        int Kenti_Jami()
        {
            int jami = 0;
            for ( int ind = 0 ; ind < masivi_1.Length ; ind++ )
                if ( masivi_1[ind] % 2 == 1 ) jami += masivi_1[ind];
            return jami;
        }
        public static int StatikuriMetodi(ChemiKlasi_2 obieqti)
        {
            return obieqti.Kenti_Jami();
        }
    }
private void button2_Click(object sender, EventArgs e)
{
array <int>^ masivi = gcnew array<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
ChemiKlasi^ obj1 = gcnew ChemiKlasi(masivi);
int shedegi = ChemiKlasi::statikuri_Funqcia(obj1);
label3->Text = shedegi.ToString();
}
```

შემთხვევითი რიცხვების გენერატორი

6.6.4.

```
{
        System::Random^ obieqti = gcnew System::Random();
        int min_rixcvi = Convert::ToInt32(textBox1->Text),          //      1
        max_ricxvi = Convert::ToInt32(textBox2->Text),              //      30
        ramdenjer = Convert::ToInt32(textBox3->Text),               //      7
        mocemuli_ricxvi = Convert::ToInt32(textBox4->Text),         //      10
        raodenoba = 0, ricxvi,
```

```
        ricxvebis_raodenoba = 0;                //      გენერირებული რიცხვების რაოდენობა

        for ( ; ; )
        {
           ricxvi = obieqti->Next(min_rixcvi, max_ricxvi);
           ricxvebis_raodenoba++;
           if ( ricxvi == mocemuli_ricxvi ) raodenoba++;
           if (raodenoba == ramdenjer) break;
        }
        label1->Text = ricxvebis_raodenoba.ToString();
    }
```

6.6.5.

```
{
        label1->Text = "";
        array<int>^ masivi = gcnew array<int> { 20, 20, 20, 20, 20 };
        int shemtxveviti_ricxvi, asantebis_raodenoba = 0;
        System::Random^ obieqti = gcnew System::Random();

        for ( ; ; )
        {
           shemtxveviti_ricxvi = obieqti->Next(5);
           masivi[shemtxveviti_ricxvi]--;
           asantebis_raodenoba++;
           if (masivi[shemtxveviti_ricxvi] == 0) break;
        }
        label1->Text = L"უნდა ამოვიღოთ  " + asantebis_raodenoba.ToString() +
                        L"  ასანთი იმისათვის, \nრომ ერთ-ერთი კოლოფი დაცარიელდეს";
    }
```

თავი 7. პოლიმორფიზმი

მეთოდის გადატვირთვა. პოლიმორფიზმი

7.1.2.

```
ref class Figura
{
public : int Perimetri(int par1)
{
return 4 * par1;
}
int Perimetri(int par1, int par2)
{
return 2 * (par1 + par2);
}
int Perimetri(int par1, int par2, int par3)
{
return par1 + par2 + par3;
```

```
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
Figura^ obieqti_1 = gcnew Figura();
int gverdi_1 = Convert::ToInt32(textBox1->Text);
int gverdi_2 = Convert::ToInt32(textBox2->Text);
int gverdi_3 = Convert::ToInt32(textBox3->Text);

int kvadratis_perimetri = obieqti_1->Perimetri(gverdi_1);
int otxkutxedis_perimetri = obieqti_1->Perimetri(gverdi_1, gverdi_2);
int samkutxedis_perimetri = obieqti_1->Perimetri(gverdi_1, gverdi_2, gverdi_3);

label1->Text = L"კვადრატის პერიმეტრი = " + kvadratis_perimetri.ToString();
label1->Text += L"\nოთხკუთხედის პერიმეტრი = " + otxkutxedis_perimetri.ToString();
label1->Text += L"\nსამკუთხედის პერიმეტრი = " + samkutxedis_perimetri.ToString();
}
```

7.2.3.

```
ref class Avtomobili
{
public : int Metodi_1(int par1, int par2)
{
return par1 * par2;
}
double Metodi_1(double par1, double par2)
{
return par1 * par2;
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
Avtomobili^ obieqti_1 = gcnew Avtomobili();
int bakis_tevadoba = Convert::ToInt32(textBox1->Text);
int manzili_1_litri = Convert::ToInt32(textBox2->Text);
double maqsimaluri_sichqare = Convert::ToDouble(textBox3->Text);
double dro = Convert::ToDouble(textBox4->Text);

int manzili_savse_baki = obieqti_1->Metodi_1(bakis_tevadoba, manzili_1_litri);
double manZili_dro = obieqti_1->Metodi_1(dro, maqsimaluri_sichqare);

label1->Text = L"სავსე ავზით გავლილი მანძილი = " + manzili_savse_baki.ToString();
label2->Text = L"მაქსიმალური სიჩქარით მოძრაობისას გავლილი მანძილი = " +
                manZili_dro.ToString();
}
```

კონსტრუქტორის გადატვირთვა

7.3.1.

```
ref class ChemiKlasi
    {
       int Min;
        public : ChemiKlasi(array<int>^ masivi1)
       {
          Min = masivi1[0];
          for ( int ind = 1; ind < masivi1->Length; ind++ )
             if ( masivi1[ind] < Min ) Min = masivi1[ind];
       }
        public : ChemiKlasi(ChemiKlasi^ obj)
       {
          Min = obj->Min;
       }
        public : void Naxva(System::Windows::Forms::Label^ lab)
       {
          lab->Text = Min.ToString();
       }
    };
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
    {
       array<int>^ masivi1 = gcnew array<int> { 6, 2, 9, 4, -7, 1 };
       ChemiKlasi^ obieqti1 = gcnew ChemiKlasi(masivi1);
       ChemiKlasi^ obieqti2 = gcnew ChemiKlasi(obieqti1);
       obieqti2->Naxva(label1);
    }
```

7.3.2.

```
ref class Chemi_Klasi
    {
       int gverdi1;
       int gverdi2;
       int gverdi3;
       int perimetri;
       double fartobi;
        public : Chemi_Klasi(int par1)
       {
          gverdi1 = par1;
          perimetri = 4 * par1;
            fartobi = gverdi1 * gverdi1;
       }
        public : Chemi_Klasi(int par1, int par2)
       {
          gverdi1 = par1;
          gverdi2 = par2;
          perimetri = 2 * ( par1 + par2 );
```

```
            fartobi = par1 * par2;
        }
         public : Chemi_Klasi(int par1, int par2, int par3)
        {
            gverdi1 = par1;
            gverdi2 = par2;
            gverdi3 = par3;
            perimetri = par1 + par2 + par3;
            fartobi = ( par1 * par2 ) / 2;
        }
         public : void Naxva(System::Windows::Forms::Label^ lab)
        {
            lab->Text = L"ფიგურის პერიმეტრი = " + perimetri.ToString() +
                        L"\nფიგურის ფართობი = " + fartobi.ToString();
        }
    };
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
          int gverdi1 = Convert::ToInt32(textBox1->Text);
          int gverdi2 = Convert::ToInt32(textBox2->Text);
          int gverdi3 = Convert::ToInt32(textBox3->Text);
         Chemi_Klasi^ obieqti1 = gcnew Chemi_Klasi(gverdi1);
         Chemi_Klasi^ obieqti2 = gcnew Chemi_Klasi(gverdi1, gverdi2);
         Chemi_Klasi^ obieqti3 = gcnew Chemi_Klasi(gverdi1, gverdi2, gverdi3);
         obieqti1->Naxva(label1);
         obieqti2->Naxva(label2);
         obieqti3->Naxva(label3);
    }
```

თავი 8. მემკვიდრეობითობა

კლასი. მემკვიდრეობითობა. protected მოდიფიკატორი

8.1.2.

```
ref class Martkutxedi
    {
        protected : int fudze;
    };
ref class Memkvidre : Martkutxedi
{
int simagle;
public : Memkvidre()
{
            simagle = 0;
          }
          Memkvidre(int par1, int par2)
          {
            simagle = par2;
```

```
            fudze = par1;
          }
          int Fartobi()
          {
            return fudze * simagle;
          }
        };
//
    private void button1_Click(object sender, EventArgs e)
    {
        int ricxvi1 = Convert::ToInt32(textBox1->Text);
        int ricxvi2 = Convert::ToInt32(textBox2->Text);
        Memkvidre^ obj2 = gcnew Memkvidre(ricxvi1, ricxvi2);
        int shedegi = obj2->Fartobi();
        label1->Text = shedegi.ToString();
    }
```

8.1.3.

```
ref class Martkutxedi_1 {
        protected : int fudze;
        public : Martkutxedi_1()
        {
                fudze = 0;
        }
        Martkutxedi_1(int par1)
        {
                fudze = par1;
        }
        void Naxva(System::Windows::Forms::Label^ lab1)
        {
                lab1->Text = fudze.ToString();
        }
};
ref class Memkvidre_1 : Martkutxedi_1
{
        int simagle;
        public : Memkvidre_1()
        {
                simagle = 0;
        }
        Memkvidre_1(int par1, int par2)
        {
                simagle = par2;
                fudze = par1;
        }
        int Fartobi()
        {
                return fudze * simagle;
```

```
        }
};
private void button2_Click(object sender, EventArgs e)
{
        int ricxvi1 = Convert::ToInt32(textBox1->Text);
        int ricxvi2 = Convert::ToInt32(textBox2->Text);
        Martkutxedi_1^ obj1 = gcnew Martkutxedi_1(ricxvi1);
        obj1->Naxva(label2);
        Memkvidre_1^ obj2 = gcnew Memkvidre_1(ricxvi1, ricxvi2);
        int shedegi = obj2->Fartobi();
        label1->Text = shedegi.ToString();
}
```

ვირტუალური ფუნქცია

8.2.1.

```
ref class Figura
    {
        public : int gverdi1, gverdi2, gverdi3, gverdi4;
        Figura() { }
        Figura(int par1, int par2, int par3, int par4) {
        gverdi1 = par1;
        gverdi2 = par2;
        gverdi3 = par3;
        gverdi4 = par4;
       }
       virtual int Perimetri() {
          return gverdi1 + gverdi2 + gverdi3 + gverdi4;
       }
    };
    ref class Otxkutxedi : public Figura {
                public : virtual int Perimetri() override {
          return ( gverdi1 + gverdi2 ) * 2;
       }
    };
//
    private void button1_Click(object sender, EventArgs e)
    {
        int gverdi1 = Convert::ToInt32(textBox1->Text);
        int gverdi2 = Convert::ToInt32(textBox2->Text);
        int gverdi3 = Convert::ToInt32(textBox3->Text);
        int gverdi4 = Convert::ToInt32(textBox4->Text);
         Otxkutxedi^ obj_memkvidre = gcnew Otxkutxedi();
        obj_memkvidre->gverdi1 = gverdi1;
        obj_memkvidre->gverdi2 = gverdi2;
        int shedegi = obj_memkvidre->Perimetri();
        label1->Text = shedegi.ToString();
        Figura^ obj_WInapari = gcnew Figura(gverdi1, gverdi2, gverdi3, gverdi4);
```

```
        int shedegi1 = obj_WInapari->Perimetri();
        label2->Text = shedegi1.ToString();
    }
```

8.2.4.

```
ref class Sabazo_Klasi
    {
        public : virtual int Metodi1(array<int,2>^ masivi1) {
         int min = 0;
         for ( int striqoni = 0 ; striqoni < 4 ; striqoni++ )
            for ( int sveti = 0 ; sveti < 4 ; sveti++ )
               if ( masivi1[striqoni, sveti] < 0 )
                {
                   min = masivi1[striqoni, sveti];
                   goto M1;
                }
           M1:
           for ( int striqoni = 0; striqoni < 4; striqoni++ )
              for ( int sveti = 0; sveti < 4; sveti++ )
                 if ( ( masivi1[striqoni, sveti] < min ) && ( masivi1[striqoni, sveti] < 0 ) )
                           min = masivi1[striqoni, sveti];
           return min;
        }
    };
    ref class Memkvidre_Klasi : public Sabazo_Klasi
    {
        public : virtual int Metodi1(array<int,2>^ masivi1) override {
         int max = 0;
         for ( int striqoni = 0 ; striqoni < 4 ; striqoni++ )
            for ( int sveti = 0 ; sveti < 4 ; sveti++ )
               if ( masivi1[striqoni, sveti] < 0 )
                {
                   max = masivi1[striqoni, sveti];
                   goto M1;
                }
        M1:
           for ( int striqoni = 0; striqoni < 4; striqoni++ )
              for ( int sveti = 0; sveti < 4; sveti++ )
                 if ( ( masivi1[striqoni, sveti] > max ) && ( masivi1[striqoni, sveti] < 0 ) )
                           max = masivi1[striqoni, sveti];
           return max;
        }
    };
//
    private void button2_Click(object sender, EventArgs e)
        array<int,2>^ masivi = gcnew array<int,2> (4, 4)
                                    { { 1, -2, -3, 4 }, { -5, 6, 7, -8 }, { 9, 10, -11, -12 }, { -13, -14, 15, 16 } };
        Sabazo_Klasi^ obj_Sabazo = gcnew Sabazo_Klasi();
```

```
        Memkvidre_Klasi^ obj_Memkvidre = gcnew Memkvidre_Klasi();
        int min = obj_Sabazo->Metodi1(masivi);
        int max = obj_Memkvidre->Metodi1(masivi);
        label1->Text = min.ToString();
        label2->Text = max.ToString();
    }
```

აბსტრაქტული კლასი და ფუნქცია

8.3.3.

```
ref class Televizori abstract {
        public : virtual int Metodi() abstract;
    };
    ref class Memkvidre_1 : public Televizori
    {
        int Wati_Erti_Saati;
        int Namushevari_Saatebi;
        public : Memkvidre_1(int par1, int par2)
        {
            Wati_Erti_Saati = par1;
            Namushevari_Saatebi = par2;
        }
        virtual int Metodi() override {
            return Wati_Erti_Saati * Namushevari_Saatebi;
        }
    };
//
    private void button1_Click(object sender, EventArgs e)
    {
        int wati = Convert::ToInt32(textBox1->Text);
        int saati = Convert::ToInt32(textBox2->Text);
        Memkvidre_1^ obj_Memkvidre = gcnew Memkvidre_1(wati, saati);
        int shedegi = obj_Memkvidre->Metodi();
        label1->Text = shedegi.ToString();
    }
```

8.3.4.

```
ref class Tanamshromeli abstract {
public : virtual int Metodi()  abstract;
    };
    ref class Memkvidre_2 : public Tanamshromeli {
        int xelfasi_erti_tvis;
                public : Memkvidre_2(int par1) {
            xelfasi_erti_tvis = par1;
        }
        virtual  int Metodi() override {
            return xelfasi_erti_tvis * 12;
        }
    };
```

```
//
    private void button2_Click(object sender, EventArgs e)
    {
        int erti_tvis_xelfasi = Convert::ToInt32(textBox1->Text);
        Memkvidre_2^ obj_memkvidre = gcnew Memkvidre_2(erti_tvis_xelfasi);
        int shedegi = obj_memkvidre->Metodi();
        label1->Text = shedegi.ToString();
    }
```

თავი 9. თვისება, ინტერფეისი, დელეგატი, მოვლენა და სახელების სივრცე

თვისება

9.1.3.

```
ref class ChemiKlasi
    {
                private : int cvladi;
       //       თვისების გამოცხადება
                public : property int ChemiTviseba
       {
          int get()
          {
             return cvladi;
          }
          void set(int par)
          {
             //   cvladi ცვლადს ენიჭება value მნიშვნელობა თუ ის 5-ის ჯერადია
             if (par % 5 == 0) cvladi = par;
          }
       }
    };
//
    private void button1_Click(object sender, EventArgs e)
    {
       ChemiKlasi^ obieqti = gcnew ChemiKlasi();
        obieqti->ChemiTviseba = Convert::ToInt32(textBox1->Text);
       label1->Text = obieqti->ChemiTviseba.ToString();
    }
```

სკალარული, ინდექსირებული და სტატიკური თვისება

9.2.1.

```
ref class Chemi_Klasi
    {
       int ricxvi1;
       int ricxvi2;
       int ricxvi3;
```

```
        int ricxvi4;
        int ricxvi5;
                public :
                        property int default[int]
                        {
                                int get(int index) { return 0; }
                                void set (int index, int par) {}
                        }
                        property int mteli[int]
        {
           int get(int index)
           {
              switch (index)
              {
                 case 0: return ricxvi1;
                 case 1: return ricxvi2;
                 case 2: return ricxvi3;
                 case 3: return ricxvi4;
                 case 4: return ricxvi5;
                 default: return 0;
              }
           }
           void set(int index, int par)
           {
              switch (index)
              {
                 case 0: this->ricxvi1 = par; break;
                 case 1: this->ricxvi2 = par; break;
                 case 2: this->ricxvi3 = par; break;
                 case 3: this->ricxvi4 = par; break;
                 case 4: this->ricxvi5 = par; break;
              }
           }
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
        label1->Text = "";
        Chemi_Klasi^ obieqti = gcnew Chemi_Klasi();
        for ( int ind = 0; ind <= 4; ind++ )
                obieqti->mteli[ind] = ind + 10;
        for (int ind = 0 ; ind <= 4 ; ind++)
                label1->Text += obieqti->mteli[ind].ToString() + "  ";
}
```

9.2.2.

```
ref class ChemiKlasi
    {
```

```
        static double ricxvi;
        public : static property double tviseba {
          double get() {
                return ricxvi;
        }
          void set(double par)
          {
                if (par <= 0) ricxvi = par;
          }
        }
    };
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
        label1->Text = "";
        ChemiKlasi^ obieqti = gcnew ChemiKlasi();
        for ( int ind = -3 ; ind <= 3 ; ind++ )
        {
                ChemiKlasi::tviseba = ind ;
                label1->Text += ChemiKlasi::tviseba.ToString() + "  ";
        }
}
```

9.2.7.

```
    class ChemiKlasi
    {
        int ricxvi1 = -1;
        int ricxvi2 = -1;
        int ricxvi3 = -1;
        int ricxvi4 = -1;
        int ricxvi5 = -1;
        public int this[int index]
        {
          get
          {
            switch ( index )
            {
              case 0: return ricxvi1;
              case 1: return ricxvi2;
              case 2: return ricxvi3;
              case 3: return ricxvi4;
              case 4: return ricxvi5;
              default: return 0;
            }
          }
          set
          {
            switch ( index )
            {
```

```
                case 0: if (value % 2 == 0) { this.ricxvi1 = value; break; } else break;
                case 1: if (value % 2 == 0) { this.ricxvi2 = value; break; } else break;
                case 2: if (value % 2 == 0) { this.ricxvi3 = value; break; } else break;
                case 3: if (value % 2 == 0) { this.ricxvi4 = value; break; } else break;
                case 4: if (value % 2 == 0) { this.ricxvi5 = value; break; } else break;
            }
          }
        }
    }
    private void button2_Click(object sender, EventArgs e)
    {
        label2.Text = "";
        ChemiKlasi obieqti = new ChemiKlasi();
        for (int ind = 0 ; ind <= 4 ; ind++)
        {
            obieqti[ind] = ind;
            label2.Text += obieqti[ind].ToString() + "  ";
        }
}
```

დელეგატი

9.3.1.

```
public delegate void Delegati(int cvladi);
//
public ref class Klasi1 {
public :
static void Funqcia1(int par) {
        par += 10;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia1 - პარამეტრი = " + par.ToString());
}
static void Funqcia2(int par) {
        par += 20;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia2 - პარამეტრი = " + par.ToString());
}
static void Funqcia3(int par) {
        par += 30;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia3 - პარამეტრი = " + par.ToString());
}
//Klasi1() : cvladi(1) { }
//Klasi1(int par) : cvladi(par) { }
//protected :
        //int cvladi;
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Delegati^ delegati1 = gcnew Delegati(Klasi1::Funqcia1);
//        delegati1 დელეგატს ემატება Funqcia2 ფუნქციის მისამართი
```

```
delegati1 += gcnew Delegati(Klasi1::Funqcia2);
delegati1(100);                 //      გამოიძახება Funqcia1 და Funqcia2 ფუნქციები ფუნქციები
//      delegati1 დელეგატიდან იშლება Funqcia1 ფუნქციის მისამართი
delegati1 -= gcnew Delegati(Klasi1::Funqcia1);
//      delegati1 დელეგატს ემატება Funqcia3 ფუნქციის მისამართი
delegati1 += gcnew Delegati(Klasi1::Funqcia3);
delegati1(200);                 //      გამოიძახება Funqcia2 და Funqcia3 ფუნქციები
}
```

9.3.2.

```
public delegate void Delegati(int cvladi);
//
ref class Klasi1 {
public :
 void Funqcia1(int par) {
        par += 10;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia1 - პარამეტრი = " + par.ToString());
}
 void Funqcia2(int par) {
        par += 20;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia2 - პარამეტრი = " + par.ToString());
}
 void Funqcia3(int par) {
        par += 30;
System::Windows::Forms::MessageBox::Show(L"მუშაობს Funqcia3 - პარამეტრი = " + par.ToString());
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi1^ obj = gcnew Klasi1;
Delegati^ delegati1 = gcnew Delegati(obj, &Klasi1::Funqcia1);
//Delegati^ delegati1 = gcnew Delegati(Klasi1::Funqcia1);
//      delegati1 დელეგატს ემატება Funqcia2 ფუნქციის მისამართი
delegati1 += gcnew Delegati(obj, &Klasi1::Funqcia2);
delegati1(100);                 //      გამოიძახება Funqcia1 და Funqcia2 ფუნქციები ფუნქციები
//      delegati1 დელეგატიდან იშლება Funqcia1 ფუნქციის მისამართი
delegati1 -= gcnew Delegati(obj, &Klasi1::Funqcia2);
//      delegati1 დელეგატს ემატება Funqcia3 ფუნქციის მისამართი
delegati1 += gcnew Delegati(obj, &Klasi1::Funqcia3);
delegati1(200);                 //      გამოიძახება Funqcia2 და Funqcia3 ფუნქციები
}
```

მოვლენა

9.4.1.

```
public delegate void Delegati2(int ricxvi1, int ricxvi2);
//      მოვლენის შემცველი კლასი
public ref class Klasi {
public :
```

```
//        მოვლენა იძახებს Delegati დელეგატთან ასოცირებულ ფუნქციებს
event Delegati2^ Movlena;
//        მოვლენის აღმძვრელი ფუნქცია
 void Funqcia_Movlena(int par1, int par2)
 {
if ( par1 > par2 ) Movlena(par1, par2);
 }
};
//        კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებელს
public ref class Klasi_Damamushavebeli2 {
public :
void Funqcia_Damamushavebeli_1(int par1, int par2)
{
System::Windows::Forms::MessageBox::Show(L"პირველი რიცხვი მეტია მეორეზე");
}
};
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj2 = gcnew Klasi;
Klasi_Damamushavebeli2^ dam2 = gcnew Klasi_Damamushavebeli2;
//
obj2->Movlena += gcnew Delegati2(dam2, &Klasi_Damamushavebeli2::Funqcia_Damamushavebeli_1);
obj2->Funqcia_Movlena(cvladi1, cvladi2);                //        Movlena მოვლენის ინიცირება
}
```

9.4.4.

```
public delegate void Delegati2(array<int>^ masivi1);
//        მოვლენის შემცველი კლასი
public ref class Klasi {
public :
//        მოვლენა იძახებს Delegati დელეგატთან ასოცირებულ ფუნქციებს
event Delegati2^ Movlena;
//        მოვლენის აღმძვრელი ფუნქცია
 void Funqcia_Movlena(array<int>^ masivi1)
 {
            int alami = 0;
            for ( int index = 0; index < masivi1->Length; index++ )
                  if ( masivi1[index] >= 50 ) alami = 1;
            if ( alami == 0 ) Movlena(masivi1);
 }
};
//        კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებელს
public ref class Klasi_Damamushavebeli2 {
public :
void Funqcia_Damamushavebeli_1(array<int>^ masivi1)
{
System::Windows::Forms::MessageBox::Show(L"მასივის ყველა ელემენტი 50-ზე ნაკლებია");
```

```
}
};
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj2 = gcnew Klasi;
array<int>^ masivi2 = gcnew array<int> { 1, 2, 3, 4, 5 };
Klasi_Damamushavebeli2^ dam2 = gcnew Klasi_Damamushavebeli2;
//
obj2->Movlena += gcnew Delegati2(dam2, &Klasi_Damamushavebeli2::Funqcia_Damamushavebeli_1);
obj2->Funqcia_Movlena(masivi2);       //          Movlena მოვლენის ინიცირება
}
```

9.4.5.

```
public delegate void Delegati2(array<int>^ masivi1);
//          მოვლენის შემცველი კლასი
public ref class Klasi {
public :
//          მოვლენა იძახებს Delegati დელეგატთან ასოცირებულ ფუნქციებს
event Delegati2^ Movlena;
//          მოვლენის აღმძვრელი ფუნქცია
void Funqcia_Movlena(array<int>^ masivi1, array<int>^ masivi2, array<int>^ masivi3)
{
          for ( int index = 0; index < masivi1->Length; index++ )
                    if ( masivi2[index] == 0 ) Movlena(masivi1);
                              else masivi3[index] = masivi1[index] / masivi2[index];
}
};
//          კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებელს
public ref class Klasi_Damamushavebeli2 {
public :
void Funqcia_Damamushavebeli_1(array<int>^ masivi1)
{
System::Windows::Forms::MessageBox::Show(L"ადგილი აქვს ნულზე გაყოფას");
}
};
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi1 = Convert::ToInt32(textBox1->Text);
int cvladi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj2 = gcnew Klasi;
array<int>^ masivi1 = gcnew array<int> { 11, 22, 33, 44, 55 };
array<int>^ masivi2 = gcnew array<int> { 1, 2, 0, 4, 5 };
array<int>^ masivi3 = gcnew array<int> (masivi1->Length) { 1, 2, 3, 4, 5 };
Klasi_Damamushavebeli2^ dam2 = gcnew Klasi_Damamushavebeli2;
//
obj2->Movlena += gcnew Delegati2(dam2, &Klasi_Damamushavebeli2::Funqcia_Damamushavebeli_1);
obj2->Funqcia_Movlena(masivi1, masivi2, masivi3);          //          Movlena მოვლენის ინიცირება
}
```

სახელების სივრცე

9.5.1.

```
namespace Sivrce_1
   {
      public ref class Samkutxedi
      {
                                    public : double Fartobi(int par1, int par2)
         {
            return par1 * par2 / 2;
         }
      };
   }
   namespace Sivrce_2
   {
      public ref class Samkutxedi
      {
                                    public : int Perimetri(int par1, int par2, int par3)
         {
            return par1 + par2 + par3;
         }
      };
   }
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
           Sivrce_1::Samkutxedi^ samkutxedi_1 = gcnew Sivrce_1::Samkutxedi();
           Sivrce_2::Samkutxedi^ samkutxedi_2 = gcnew Sivrce_2::Samkutxedi();
           int gverdi_1 = Convert::ToInt32(textBox1->Text);
           int gverdi_2 = Convert::ToInt32(textBox2->Text);
           int gverdi_3 = Convert::ToInt32(textBox3->Text);
           double fartobi = samkutxedi_1->Fartobi(gverdi_1, gverdi_2);
           int perimetri = samkutxedi_2->Perimetri(gverdi_1, gverdi_2, gverdi_3);
           label1->Text = fartobi.ToString();
           label2->Text = perimetri.ToString();
}
```

9.5.2.

```
namespace Sivrce
   {
      namespace Sivrce_1
      {
         public ref class Masivi
         {
            public : int Metodi1(array<int>^ masivi1)
            {
               int jami = 0;
               for ( int indexi = 0; indexi < masivi1->Length; indexi++ )
                  if ( masivi1[indexi] >= 0 )  jami += masivi1[indexi];
```

```
            return jami;
        }
    };
}
namespace Sivrce_2
{
    public ref class Masivi
    {
        public : int Metodi2(array<int>^ masivi1)
        {
            int namravli = 1;
            for ( int indexi = 0; indexi < masivi1->Length; indexi++ )
                if ( masivi1[indexi] < 0 ) namravli *= masivi1[indexi];
            return namravli;
        }
    };
}
}
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
        array<int>^ masivi = gcnew array<int> { 1, -2, 3, -4, 5, -6 };
        Sivrce::Sivrce_1::Masivi^ obieqti_1 = gcnew Sivrce::Sivrce_1::Masivi();
        Sivrce::Sivrce_2::Masivi^ obieqti_2 = gcnew Sivrce::Sivrce_2::Masivi();
        int jami = obieqti_1->Metodi1(masivi);
        int namravli = obieqti_2->Metodi2(masivi);
        label1->Text = jami.ToString();
        label2->Text = namravli.ToString();
}
```

ინტერფეისი. რეალიზება

9.6.1.

```
//      ინტერფეისული კლასის გამოცხადება
interface class Interpeisi {
        double Kvadrati(int par1);
        double Kubi(int par2);
};
ref class Klasi : Interpeisi {
public :
        virtual double Kvadrati(int par1) {
                return System::Math::Pow(par1, 2);
        }
        virtual double Kubi(int par2) {
                return System::Math::Pow(par2, 3);
        }
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
```

```
int ricxvi1 = Convert::ToInt32(textBox1->Text);
int ricxvi2 = Convert::ToInt32(textBox2->Text);
Klasi^ obj1 = gcnew Klasi();
double kvadrati = obj1->Kvadrati(ricxvi1);
double Kubi = obj1->Kubi(ricxvi2);
label1->Text = kvadrati.ToString();
label2->Text = Kubi.ToString();
}
```

9.6.2.

```
//      ინტერფეისული კლასის გამოცხადება
interface class Interpeisi {
        property int tviseba
                {
        int get();
        void set(int par);
}
};
ref class Klasi : Interpeisi {
        int cvladi;
public :
        property int tviseba
                {
        virtual int get()
        {
                return cvladi;
        }
        virtual void set(int par)
        {
                if ( par % 2 == 0 ) cvladi = par;
        }
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj1 = gcnew Klasi();
obj1->tviseba = Convert::ToInt32(textBox1->Text);
label1->Text = obj1->tviseba.ToString();
}
```

9.6.3.

```
//      ინტერფეისული კლასის გამოცხადება
interface class Interpeisi {
        property int tviseba[int] {
        int get(int index);
        void set(int index, int par);
}
};
ref class Klasi : Interpeisi {
```

```
        int cvladi;
public :
        property int tviseba[int] {
        virtual int get(int index)
        {
                return cvladi;
        }
        virtual void set(int index, int par)
        {
                if ( par > 20 ) cvladi = par;
                else cvladi = 0;
        }
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
Klasi^ obj1 = gcnew Klasi();
array<int>^ mas = gcnew array<int> { 10, 30, 20, 40, 50 };
for ( int ind = 0; ind < mas->Length; ind++ )
{
        obj1->tviseba[0] = mas[ind];
        label1->Text += obj1->tviseba[ind].ToString() + "    ";
}
}
```

9.6.4.

```
//      დელეგატის გამოცხადება
public delegate void Delegati2(int ricxvi);
//      ინტერფეისული კლასის გამოცხადება
interface class Interpeisi {
        event Delegati2^ Movlena;
};
public ref class Klasi : public Interpeisi {
public :
//      მოვლენა იძახებს Delegati დელეგატთან ასოცირებულ ფუნქციებს
virtual event Delegati2^ Movlena;
//      მოვლენის აღმძვრელი ფუნქცია
 void Funqcia_Movlena(int par1)
 {
if ( par1 > 25 ) Movlena(par1);
 }
};
//      კლასი შეიცავს Movlena მოვლენის ფუნქცია-დამამუშავებელს
public ref class Klasi_Damamushavebeli2 {
public :
void Funqcia_Damamushavebeli_1(int par2)
{
System::Windows::Forms::MessageBox::Show(L" აღიძვრა მოვლენა\n" +
```

```
par2.ToString() + L" > 25");
}
};
//
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
int cvladi = Convert::ToInt32(textBox1->Text);
Klasi^ obj2 = gcnew Klasi();
Klasi_Damamushavebeli2^ dam2 = gcnew Klasi_Damamushavebeli2;
//
obj2->Movlena += gcnew Delegati2(dam2, &Klasi_Damamushavebeli2::Funqcia_Damamushavebeli_1);
obj2->Funqcia_Movlena(cvladi);          //          Movlena მოვლენის ინიცირება
}
```

თავი 10. ინფორმაციის შეტანა-გამოტანა

ფაილში ბაიტების შეტანა-გამოტანა. FileStream კლასი

10.1.1.

```
{
        label1->Text = "";
        int wakitxuli_baitebis_raodenoba;
        Byte jami = 0;
        array<Byte>^ masivi = gcnew array<Byte> (5);
        FileStream^ file1 = gcnew FileStream("filetext.txt", FileMode::Open);

        wakitxuli_baitebis_raodenoba = file1->Read(masivi, 0, 5);

        for ( int indexi = 0; indexi < masivi->Length; indexi++ )
           jami += masivi[indexi];
        file1->WriteByte(jami);
        file1->Close();
        for ( int indexi = 0; indexi < masivi->Length; indexi++ )
           label1->Text += masivi[indexi].ToString() + "   ";
        label1->Text += jami.ToString();
        label2->Text = wakitxuli_baitebis_raodenoba.ToString();
}
```

10.1.2.

```
{
        label1->Text = "";
        int wakitxuli_baitebis_raodenoba;
        array<Byte>^ masivi = gcnew array<Byte> (5);
        FileStream^ file1 = gcnew FileStream("file1.txt", FileMode::Open);
        FileStream^ file2 = gcnew FileStream("file2.txt", FileMode::Create);

        wakitxuli_baitebis_raodenoba = file1->Read(masivi, 0, 5);
        file2->Write(masivi, 0, 5);
        file1->Close();
```

```
        file2->Close();
        for ( int indexi = 0; indexi < masivi.Length; indexi++ )
           label1->Text += masivi[indexi].ToString() + "   ";
        label2->Text = wakitxuli_baitebis_raodenoba.ToString();
}
```

10.1.3.

```
{
int ricxvi;
FileStream^ file_in;
FileStream^ file_out;
//        საწყისი ფაილის გახსნა
file_in = gcnew FileStream("file1.txt", FileMode::Open);
//        მიმღები ფაილის გახსნა
file_out = gcnew FileStream("file2.txt", FileMode::Create);
label1->Text = "";
//        ბაიტების გადაწერა file_in ფაილიდან file_out ფაილში
for ( int i = 1; i <= 10; i++ )
{
//        ბაიტების წაკითხვა file_in ფაილიდან
ricxvi = file_in->ReadByte();
//        ბაიტების ჩაწერა file_out ფაილში
if ( ricxvi != -1 ) file_out->WriteByte( ( Byte ) ricxvi );
label1->Text += ricxvi + "   ";
}
//        ფაილების დახურვა
file_in->Close();
file_out->Close();
}
```

ფაილში სიმბოლოების შეტანა-გამოტანა. StreamReader და StreamWriter კლასები

10.2.1.

```
{
        label1->Text = "";
        FileStream^ file_in;
        array<Char>^ simboloebis_masivi = gcnew array<Char> (14);
        file_in = gcnew FileStream("file1.txt", FileMode::Open);
        StreamReader^ file_stream_in = gcnew StreamReader(file_in);

        int wakitxuli_simboloebis_raodenoba = file_stream_in->Read(simboloebis_masivi, 0, 14);
        for ( int i = 0; i < simboloebis_masivi->Length; i++ )
             label1->Text += simboloebis_masivi[i].ToString();
        file_stream_in->Close();
         //
        FileStream^ file_out;
        file_out = gcnew FileStream("file2.txt", FileMode::Create);
        StreamWriter^ file_stream_out = gcnew StreamWriter(file_out);
        //          file2.txt ფაილში simboloebis_masivi სტრიქონის ჩაწერა
```

```
        file_stream_out->Write(simboloebis_masivi);
        file_stream_out->Close();
}
```

10.2.2.

```
{
        label1->Text = "";
        FileStream^ file_in;
        array<Char>^ simboloebis_masivi = gcnew array<Char> (14);
        file_in = gcnew FileStream("file1.txt", FileMode::Open);
        StreamReader^ file_stream_in = gcnew StreamReader(file_in);

        int wakitxuli_simboloebis_raodenoba = file_stream_in->Read(simboloebis_masivi, 0, 14);
        for ( int i = 0; i < simboloebis_masivi->Length; i++ )
           label1->Text += simboloebis_masivi[i].ToString();
        file_stream_in->Close();
        FileStream^ file_out;
        file_out = gcnew FileStream("file1.txt", FileMode::Append);
        StreamWriter^ file_stream_out = gcnew StreamWriter(file_out);
        //          file2.txt ფაილში simboloebis_masivi სტრიქონის ჩაწერა
        file_stream_out->Write(simboloebis_masivi);
        file_stream_out->Close();
}
```

10.2.3.

```
{
        label1->Text = ""; label2->Text = "";
        Char simbolo;
        FileStream^ file_in;
        file_in = gcnew FileStream("file1.txt", FileMode::Open);
        StreamReader^ file_stream_in = gcnew StreamReader(file_in);
        FileStream^ file_out;
        file_out = gcnew FileStream("file2.txt", FileMode::Create);
        StreamWriter^ file_stream_out = gcnew StreamWriter(file_out);
        for ( int i = 0; i < file_in->Length; i++ )
        {
           simbolo = ( char ) file_stream_in->Read();
           file_stream_out->Write(simbolo);
           label2->Text += ( Char ) simbolo;
        }
        file_stream_in->Close();
        file_stream_out->Close();
}
```

ფაილებთან პირდაპირი მიმართვა. Seek ფუნქცია

10.4.2.

```
{
        label1->Text = "";
        array<Byte>^ masivi1 = gcnew array<Byte> { 255, 100, 200, 1, 50, 130, 250, 20, 90, 190 };
```

```
        array<Byte>^ masivi2 = gcnew array<Byte> (masivi1->Length);
        FileStream^ file_out1 = gcnew FileStream("file3.txt", FileMode::Create);
        //      ბაიტების ჩაწერა
        file_out1->Write(masivi1, 0, 9);
        file_out1->Close();
        //      შეგვაქვს წანაცვლების მნიშვნელობა
        int seek = Convert::ToInt32(textBox1->Text);
        FileStream^ file_out2 = gcnew FileStream("file3.txt", FileMode::Open);
        file_out2->Seek(seek, SeekOrigin::Begin);
        //      ბაიტების ჩაწერა მითითებული პოზიციიდან
        file_out2->Write(masivi1, 0, 4);
        file_out2->Close();
        //      ბაიტების წაკითხვა
        FileStream^ file_in = gcnew FileStream("file3.txt", FileMode::Open);
        file_in->Read(masivi2, 0, 9);
        for ( int index = 0; index < masivi2->Length; index++ )
                label1->Text += masivi2[index].ToString() + "   ";
        file_in->Close();
}
```

10.4.4.

```
{
        label1->Text = "";
        array<Byte>^ masivi1 = gcnew array<Byte> { 255, 100, 200, 1, 50, 130, 250, 20, 90, 190 };
        array<Byte>^ masivi2 = gcnew array<Byte> (masivi1->Length);
        FileStream^ file_out1 = gcnew FileStream("file3.txt", FileMode::Create);
        //      ბაიტების ჩაწერა
        file_out1->Write(masivi1, 0, 9);
        file_out1->Close();
        //      შეგვაქვს წანაცვლების მნიშვნელობა
        int seek = Convert::ToInt32(textBox1->Text);
        FileStream^ file_in = gcnew FileStream("file3.txt", FileMode::Open);
        file_in->Seek(seek, SeekOrigin::Begin);
        //      ბაიტების წაკითხვა
        file_in->Read(masivi2, 0, 4);
        for ( int index = 0; index < masivi2->Length; index++ )
                label1->Text += masivi2[index].ToString() + "   ";
        file_in->Close();
}
```

ფაილის შესახებ ინფორმაციის მიღება. FileInfo კლასი

10.5.1.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილის ატრიბუტია: " + obieqti->Attributes.ToString() + "\n";
}
```

10.5.2.

```
{
```

```
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილის შექმნის თარიღია: " + obieqti->CreationTime.ToString() + "\n";
}
```

10.5.3.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text  = L"ფაილი მოთავსებულია კატალოგში: " +
                        obieqti->Directory->ToString() + "\n";
        label1->Text += L"ფაილი მოთავსებულია კატალოგში: " + obieqti->DirectoryName + "\n";
}
```

10.5.4.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        if ( obieqti->Exists ) label1->Text = L"ფაილი არსებობს";
                else label1->Text = L"ფაილი არ არსებობს";
}
```

10.5.5.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილის გაფართოებაა: " + obieqti->Extension + "\n";
}
```

10.5.6.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილის სრული სახელია: " + obieqti->FullName + "\n";
}
```

10.5.7.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილთან უკანასკნელი მიმართვის დროა: " +
                        obieqti->LastAccessTime.ToString() + "\n";
}
```

10.5.8.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილში უკანასკნელი ჩაწერის დროა: " +
                        obieqti->LastWriteTime.ToString() + "\n";
}
```

10.5.9.

```
{
        FileInfo^ obieqti = gcnew FileInfo("file1.txt");
        label1->Text = L"ფაილის სიგრძეა: " + obieqti->Length.ToString() + "\n";
}
```

კატალოგებთან მუშაობა. DirectoryInfo კლასი

10.6.2.

```
{
```

```
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        label1->Text = L"კატალოგის შექმნის დროა: " + obj1->CreationTime.ToString() + "\n";
}
```

10.6.3.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        label1->Text = L"კატალოგის სრული სახელია: " + obj1->FullName->ToString() + "\n";
}
```

10.6.4.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        label1->Text = L"კატალოგთან უკანასკნელი მიმართვის დროა: " +
                        obj1->LastAccessTime.ToString() + "\n";
}
```

10.6.5.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        label1->Text = L"კატალოგში უკანასკნელი ჩაწერის დროა: " +
                        obj1->LastWriteTime.ToString() + "\n";
}
```

10.6.6.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        if ( obj1->Exists ) label1->Text = L"კატალოგი არსებობს";
                else label1->Text = L"კატალოგი არ არსებობს";
}
```

10.6.7.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("Install");
        label1->Text = L"მშობელი კატალოგია: " + obj1->Parent->ToString() + "\n";
}
```

10.6.8.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install");
        label1->Text = L"ძირითადი კატალოგია: " + obj1->Root->ToString() + "\n";
}
```

10.6.9.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Windows");
        array<DirectoryInfo^>^ dir = obj1->GetDirectories();
        for each ( DirectoryInfo^ x in dir )
                label1->Text += x->ToString() + "\n";
}
```

10.6.10.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Windows");
        array<FileInfo^>^ dir = obj1->GetFiles();
```

```
        for ( int ind = 0; ind < dir->Length; ind++ )
                label1->Text += dir[ind]->ToString() + "\n";
}
```

10.6.11.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Katalogi_1");
        //      Katalogi_2 კატალოგი არ უნდა არსებობდეს
        obj1->MoveTo("C:\\Katalogi_2");
}
```

10.6.12.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install_1");
        //      C: დისკზე იქმნება Install1 კატალოგი
        obj1->Create();
}
```

10.6.13.

```
{
        DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install_1");
        //      Install1 კატალოგში იქმნება Install2 ქვეკატალოგი
        obj1->CreateSubdirectory("Install_2");
}
```

10.6.14.

```
{
         DirectoryInfo^ obj1 = gcnew DirectoryInfo("C:\\Install_1");
         //      Install_1 კატალოგის წაშლა. Install_1 კატალოგი ცარიელი უნდა იყოს
         obj1->Delete();
}
```

თავი 11. განსაკუთრებული სიტუაციები

განსაკუთრებული სიტუაციები. try, catch და throw ოპერატორები

11.1.1.

```
{
        array<int>^ masivi = gcnew array<int> { 1, 2, 3, 4, 5 };
        int index, jami = 0;
        try
        {
           for ( index = 0; index <= masivi->Length; index++ )
              jami += masivi[index];
        }
        catch ( IndexOutOfRangeException^ )
        {
           jami = 0;
           label2->Text = L"ინდექსი გადის დასაშვები მნიშვნელობების დიაპაზონის გარეთ";
        }
        label1->Text = jami.ToString();
```

```
}
```

11.1.2.

```
{
        array<int, 2>^ masivi = gcnew array<int, 2> (3,3) { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
        int striqoni, sveti, jami = 0;
        try
        {
           for ( striqoni = 0; striqoni < 4; striqoni++ )
              for ( sveti = 0; sveti < 3; sveti++ )
                 jami += masivi[striqoni,sveti];
        }
        catch ( IndexOutOfRangeException^ )
        {
           jami = 0;
           label2->Text = L"ინდექსი გადის დასაშვები მნიშვნელობების დიაპაზონის გარეთ";
        }
        label1->Text = jami.ToString();
}
```

11.1.3.

```
{
        array<int>^ masivi1 = gcnew array<int> { 10, 20, 30, 40, 50 };
        array<int>^ masivi2 = gcnew array<int> { 1, 0, 3, 0, 5 };
        array<int>^ shedegi = gcnew array<int> (masivi1->Length);
        try
        {
           for ( int index = 0; index < masivi1->Length; index++ )
                 shedegi[index] = masivi1[index] / masivi2[index];
        }
        catch ( DivideByZeroException^ )
        {
                 label2->Text = L"ადგილი აქვს 0-ზე გაყოფას";
        }
        for ( int index = 0; index < masivi1->Length; index++ )
                 label1->Text += shedegi[index].ToString() + "  ";
}
```

11.1.4.

```
{
         array<int>^ masivi = gcnew array<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
         int indexi;
         BinaryWriter^ file_out;
         try
         {
         file_out = gcnew BinaryWriter(gcnew FileStream("file1.txt", FileMode::Create));
         for ( indexi = 0; indexi < masivi->Length; indexi++ )
              file_out->Write(masivi[indexi]);
         }
         catch ( IOException^ arg1 )
```

```
        {
                label1->Text = arg1->Message;
        }
        file_out->Close();
}
```

11.1.5.

```
{
        array<int, 2>^ masivi = gcnew array<int, 2> (3,3);
        int striqoni, sveti;
        BinaryReader^ file_in;
        try
        {
        file_in = gcnew BinaryReader(gcnew FileStream("file1.txt", FileMode::Open));
        for ( striqoni = 0; striqoni < 3; striqoni++ )
        {
                for ( sveti = 0; sveti < 3; sveti++ )
                {
                        masivi[striqoni, sveti] = file_in->ReadInt32();
                        label1->Text += masivi[striqoni, sveti].ToString() + "  ";
                }
                label1->Text += "\n";
        }
        }
        catch ( IOException^ arg1 )
        {
           label2->Text = arg1->Message;
        }
}
```

ლიტერატურა

1. რ. სამხარაძე. Visual C#.NET. საქართველოს ტექნიკური უნივერსიტეტი. 2009. - 455 გვ.
2. რ. სამხარაძე. SQL სერვერი. საქართველოს ტექნიკური უნივერსიტეტი. 2008. - 374 გვ.
3. Шилдт. Г. Самоучитель C++ : Пер. с англ. СПб. 2001. – 688 с.
4. Хортон А. Visual C++ 2005. Базовый курс.
5. Юлин В., Булатова. И. Приглашение к СИ. – Мн.: Выш. шк., 1990. – 224 с.
6. Галявов И. Borland C++ для себя. – М.: ДМК Пресс, 2001. – 432 с.
7. Керниган Б., Ричи Д. Язык С.
8. Лаптев В. C++. Экспресс курс. СПб.: БХВ-Петербург, 2004. - 512 с.
9. Липпман. C++ для начинающих.
10. Франка П. C++: учебный курс. — СПб.: Питер, 2003. — 521 с.
11. Культин Н. C/C++ в задачах и примерах. — СПб.: БХВ-Петербург, 2005. - 288 с.
12. Каррано Ф., Причард Дж. Абстракция данных и решение задач на C+-I-. Стены и зеркала, - издание. : Пер. с англ. — М.: Издательский дом "Вильяме", 2003. — 848 с .
13. Штерн В. Основы C++: Методы программной инженерии.
14. Дж. Сик, Л. Ли, Э. Ламсдэйн. C++ Boost Graph Library. Библиотека программиста / Пер. английского Сузи Р. - СПб.: Питер, 2006. — 304 с.
15. Элджер Д. C++ .
16. Якушев Д. «Философия» программирования на языке C++. - М.: Бук_пресс, 2006. — 320 с.
17. Рассохин Д. От Си к Си++.
18. Романов Е. Практикум по программированию на C++: Уч. пособие. СПб: БХВ-Петербург, 2004. - 432 с.
19. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/ Поиск: Пер. с англ./Роберт Седжвик. - К.: Издательство «ДиаСофт», 2001.- 688 с.
20. Прайс дж., Гандерлой М. Visual C#.NET .
21. O'Reilly. Programming C# .
22. Г. Дейтел. Введение в операционные системы. В 2-х томах. Пер. с англ. - М.: Мир, 1987.
23. Э. Таненбаум. Современные операционные системы. - СПб.: Питер, 2002. - 1040 с.: ил.
24. E.Butow, T. Ryan. Your visual blueprint for building .NET applications.
25. Harold Davis. Visual C# .NET Programming.
26. A. Hejlsberg, S.Wiltamuth. C# Language Reference.
27. Г. Шилдт. Полный справочник по C#.

რედაქტორი მ. ბაზაძე

გადაეცა წარმოებას 15.11.2011. ხელმოწერილია დასაბეჭდად 20.01.2012. ქაღალდის ზომა 60X84 1/8. პირობითი ნაბეჭდი თაბახი 30. ტირაჟი 50 ეგზ.

საგამომცემლო სახლი "ტექნიკური უნივერსიტეტი", თბილისი, კოსტავას 77