თავი 2. C++ ენის საფუძვლები

პირველი პროგრამა

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

```
{
//      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. ცვლადის სახელი არის მეხსიერების იმ უბნის სახელი, რომელიც ამ ცვლადისთვის გამოიყო და რომელშიც ამ ცვლადის მნიშვნელობა იქნება მოთავსებული. int სიტყვა (integer - მთელი) იწყებს ცვლადების გამოცხადებას და მიუთითებს, რომ ისინი მთელი რიცხვებია. მას მოსდევს ცვლადების სახელები ანუ იდენტიფიკატორები, რომლებიც ერთმანეთისგან მძიმეებით გამოიყოფა. წერტილ-მძიმე ამთავრებს ცვლადების გამოცხადებას, აგრეთვე ერთმანეთისგან გამყოფს ოპერატორებსა და ფუნქციებს (ფუნქცია არის სახელის მქონე 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 (ricxvi1 ცვლადისთვის გამოყოფილ უბანში ჩაიწერება მნიშვნელობა - 2). იგივე ეხება პროგრამის მომდევნო სტრიქონს. უკანასკნელ სტრიქონში jami ცვლადს ენიჭება ricxvi1 და ricxvi2 ცვლადების მნიშვნელობების ჯამი.

ზოგად შემთხვევაში, მინიჭების ოპერატორის ("=") მარჯვნივ მოთავსებული გამოსახულების ტიპი დაყვანილი უნდა იყოს მინიჭების ოპერატორის მარცხნივ მოთავსებული ცვლადის ტიპზე.

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

ერთ-ერთი ნაკლი, რომელიც ამ პროგრამას აქვს ის არის, რომ ის ახდენს მხოლოდ 2-სა და 3-ის შეკრებას. გარდა ამისა, პროგრამას შედეგი ეკრანზე არ გამოაქვს. ამ ნაკლის აღმოსაფხვრელად გამოვიყენებთ მონაცემების შეტანისა და გამოტანის ფუნქციებს. კერძოდ, მონაცემების შესატანად გამოვიყენებთ cin ფუნქციას, გამოსატანად კი - cout ფუნქციას. მათი გამოყენებით 2.1 პროგრამა შემდეგ სახეს მიიღებს:

```
{
```

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

        cin >> ricxvi1 >> ricxvi2;
        jami = ricxvi1 + ricxvi2;
        cout << "jami = " << jami << endl;
}
```

როგორც პროგრამიდან ჩანს, cin ფუნქციით ხდება ორი მთელი რიცხვის მნიშვნელობის შეტანა. პირველი რიცხვის მნიშვნელობა მიენიჭება ricxvi1 ცვლადს, მეორე რიცხვის მნიშვნელობა კი - ricxvi2 ცვლადს. მომდევნო სტრიქონში jami ცვლადს მიენიჭება ricxvi1 და ricxvi2 ცვლადების ჯამი. cout ფუნქციას ეკრანზე ჯერ გამოაქვს ბრჭყალებში მოთავსებული სტრიქონი, შემდეგ კი ჯამის მნიშვნელობა.

ახლა შევასრულოთ ეს პროგრამა. ამისათვის, ჯერ გავუშვათ Microsoft Visual Studio .NET 2013 სისტემა. გაიხსნება Microsoft Development Environment ფანჯარა (ნახ. 2.1). Start განყოფილებაში ვაჭერთ New Project მიმართვას. გაიხსნება New Project ფანჯარა (ნახ. 2.2). Project Types სიაში მოვნიშნოთ Visual C++ ელემენტი. Templates ზონაში მოვნიშნოთ Win32 Console Application ელემენტი. Name ველში უნდა შევიტანოთ პროექტის სახელი, მაგალითად, OriRicxvisShekreba. შემდეგ დავაჭიროთ Browse კლავიშს და მოვნიშნოთ ის კატალოგი, რომელ-შიც 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 2013 კატალოგის Projects ქვეკატალოგში. ეკრანზე გამოჩნდება Win32 Application Wizard ფანჯარა (ნახ. 2.3). ვაჭერთ Finish კლავიშს.

გაიხსნება პროგრამის კოდის ზონა (ნახ. 2.4). პროგრამა 2.2 უნდა შევიტანოთ return 0; ოპერატორის წინ (ნახ. 2.5). პროგრამის ბოლოს უნდა შევიტანოთ აგრეთვე, system("pause"); ფუნქცია. ეს ფუნქცია ეკრანზე დააყოვნებს ფანჯარას, რომელშიც შედეგები ჩანს. თუ ამ ფანჯარას (ნახ. 2.4) ყურადღებით დავათვალიერებთ, შევნიშნავთ, რომ Visual C++.NET სისტემა პროგრამის კოდის ნაწილს ავტომატურად ქმნის.

stdafx.h ფაილი ავტომატურად გენერირდება, როცა ვიწყებთ პროექტის შექმნას (AFX, Application Framework eXtensions). ეს ფაილი აღწერს სტანდარტულ სისტემურ და პროექტში ჩართულ ფაილებს.

პროგრამის შეტანის შემდეგ მის შესანახად უნდა დავაჭიროთ ინსტრუმენტების პანელის Save All კლავიშს ან კლავიატურის Ctrl+S კლავიშებს. რადგან კომენტარი ქართული შრიფტით გვაქვს შეტანილი, ამიტომ გაიხსნება ფანჯარა, რომელშიც უნდა ჩავრთოთ Apply to all documents გადამრთველი (ნახ. 2.6). პროგრამის შესასრულებლად გამოიყენება F5 კლავიში. თუ პროგრამა უშეცდომოდ შევიტანეთ, მაშინ ეკრანზე გაიხსნება ფანჯარა (ნახ. 2.7), რომელშიც შეგვაქვს მთელი რიცხვები: 1 და 2. ისინი ერთმანეთისგან ინტერვალით უნდა გამოვყოთ. Enter კლავიშზე დაჭერის შემდეგ დავინახავთ შედეგს. ფანჯრის დასახურად ვაჭერთ ფორმის ზედა მარჯვენა კუთხეში მოთავსებულ ☒ კლავიშს ან კლავიატურის Esc კლავიშს.

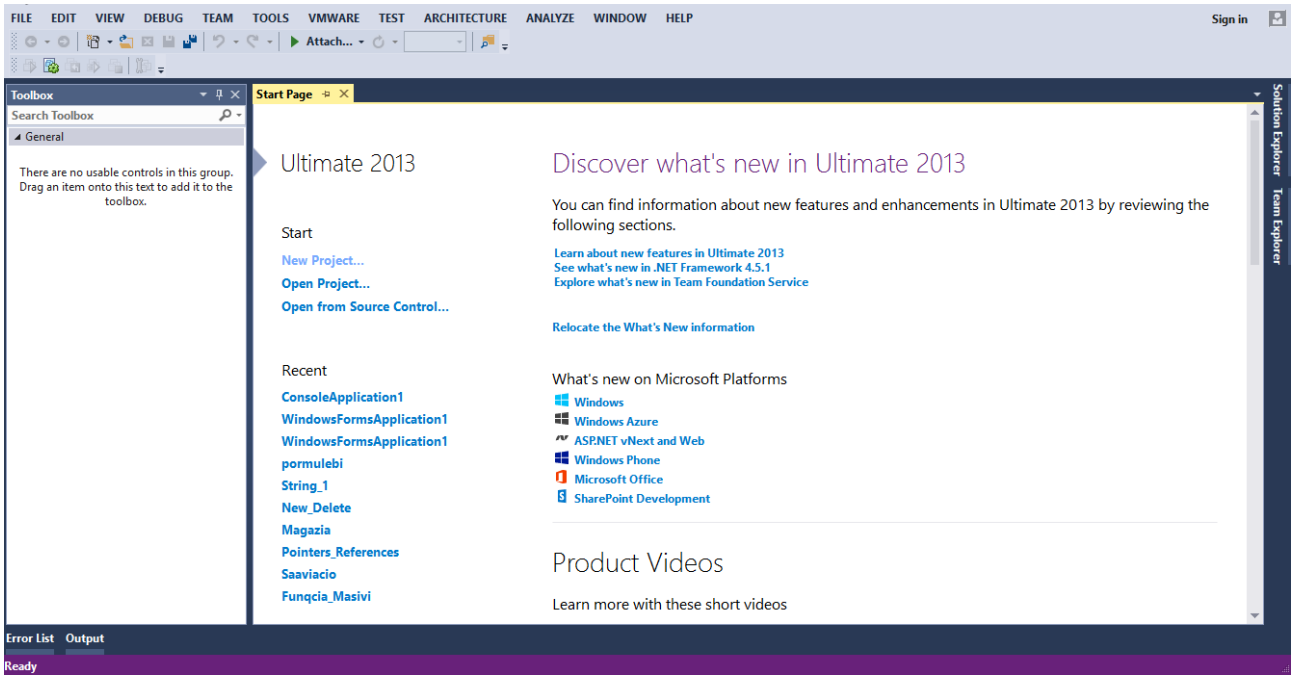
C++ პროგრამის ფაილებს აქვთ .h და .cpp გაფართოება (ტიპი). მაგალითად, იმ ფაილის სახელი, რომელშიც ჩვენი პროგრამის საწყისი კოდი შეგვაქვს, არის OriRicxvisShekreba.cpp .

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

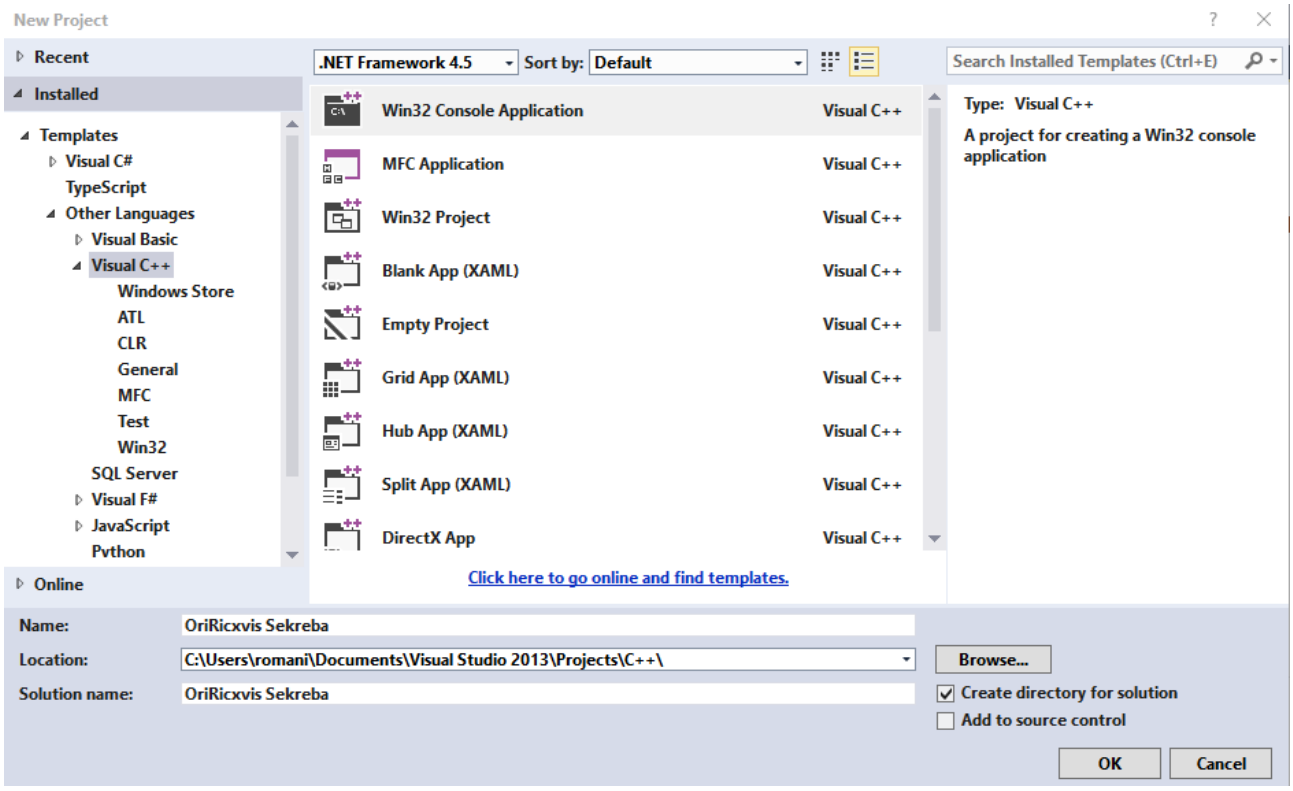
```
{
//      C++ ენაში განსხვავებულია ზედა და ქვედა რეგისტრის სიმბოლოები
int ricxvi1;
```

Ricxvi1 = 5; // შეცდომა! Ricxvi1 ცვლადი არ არის გამოცხადებული
}

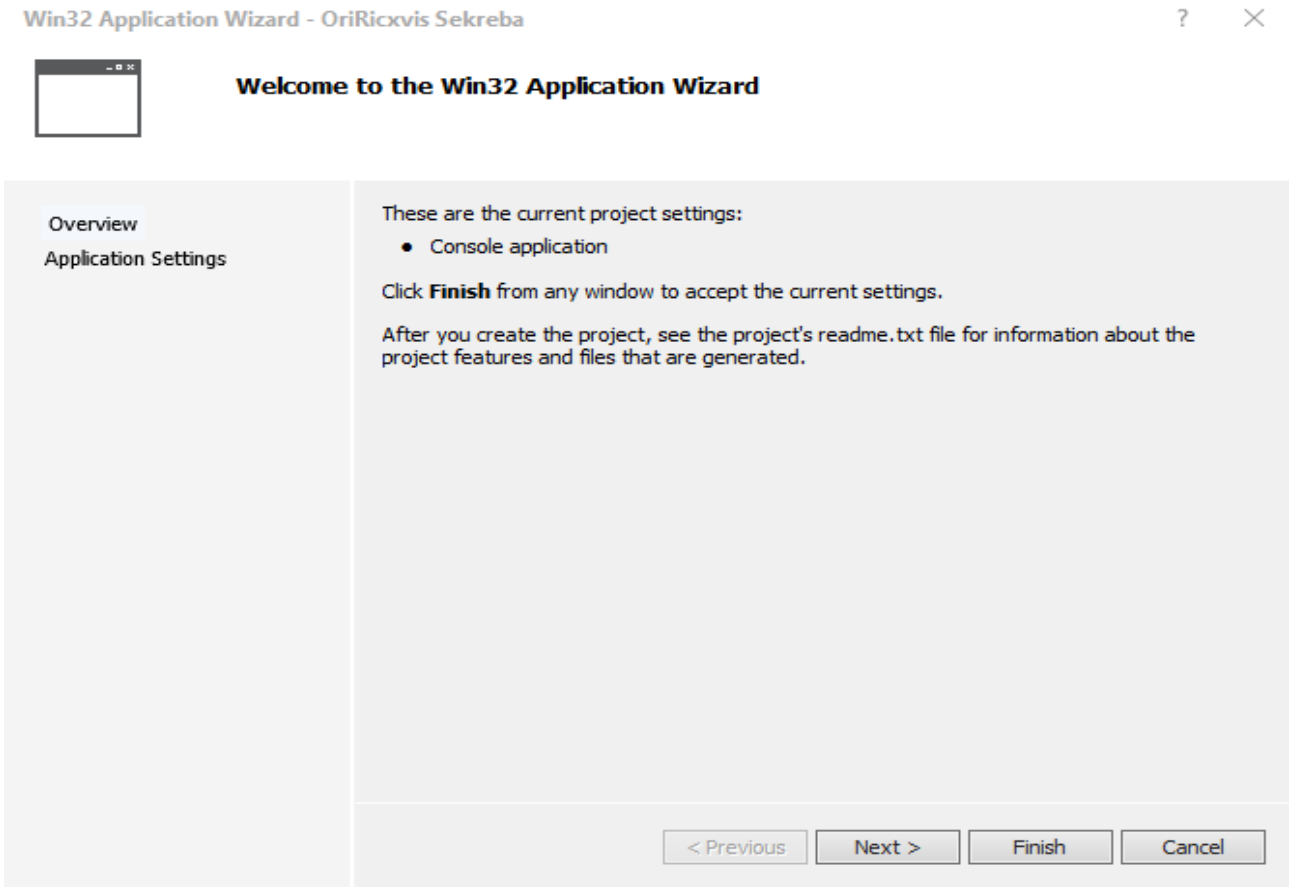
როგორც მაგალითიდან ჩანს, გამოცხადებული იყო ცვლადი, რომლის სახელია ricxcvi1, ხოლო გამოიყენება სახელი Ricxvi1. C++ ენისთვის ეს ორი სხვადასხვა სახელია.



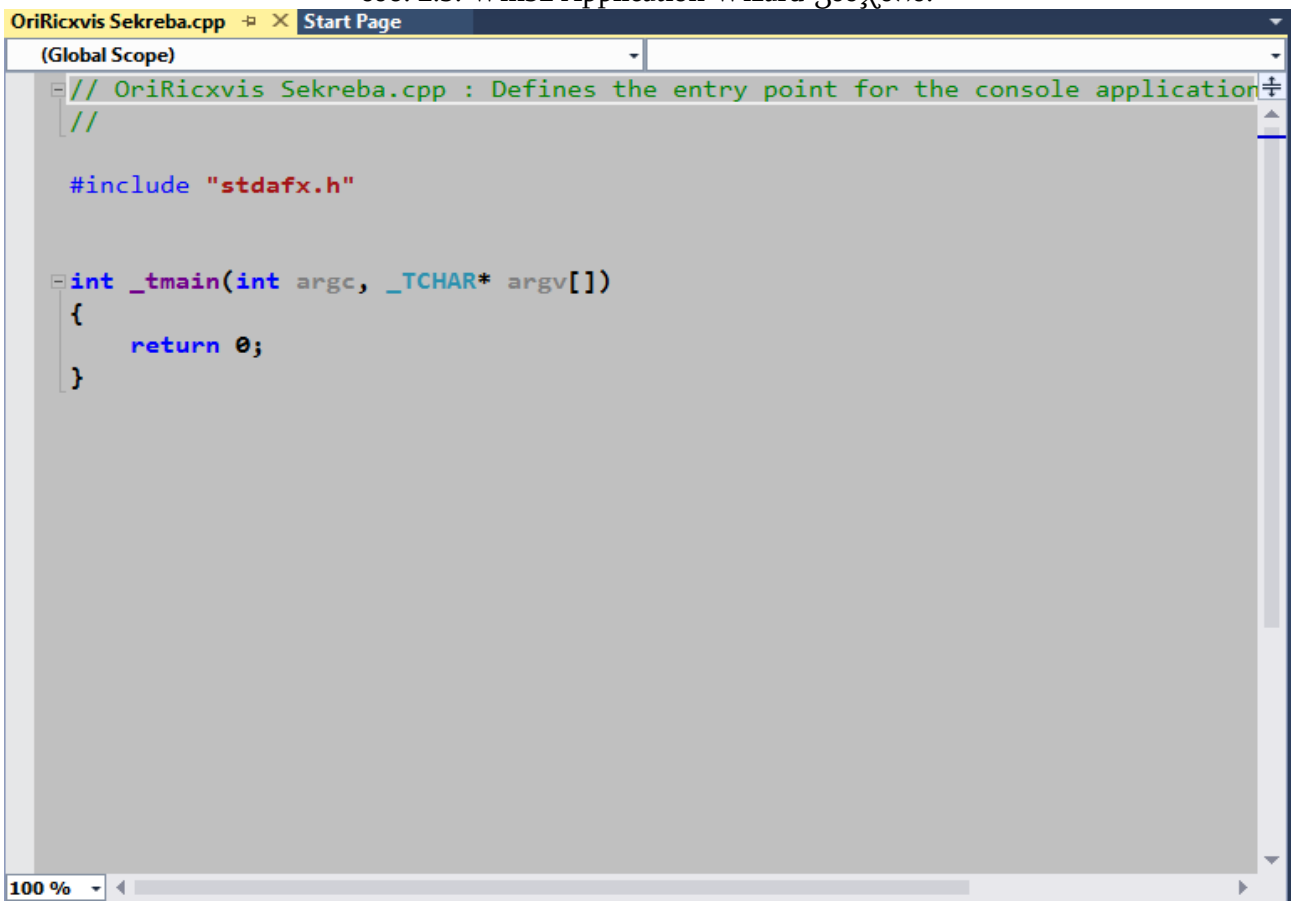
ნახ. 2.1. Start Page ფანჯარა.



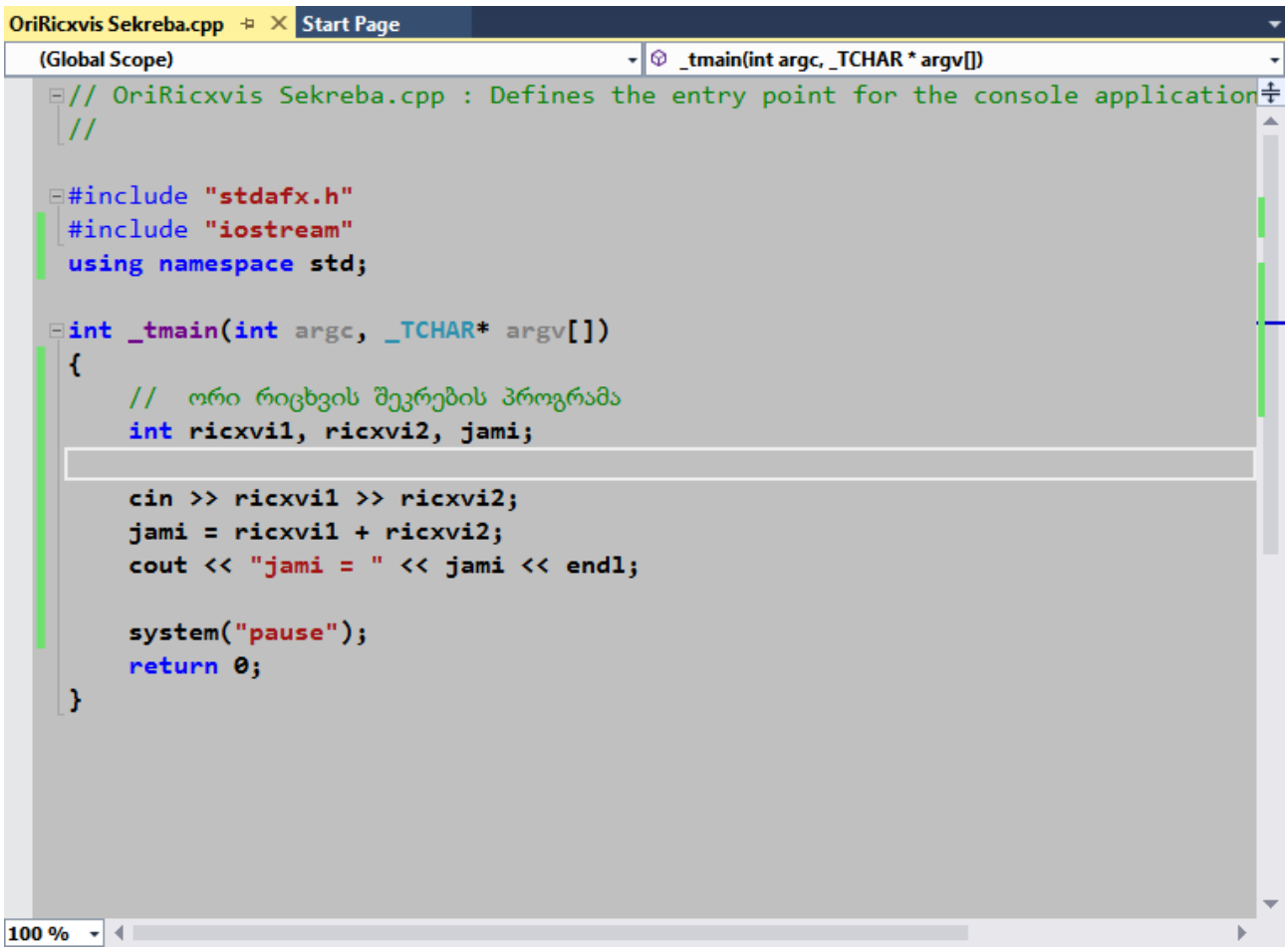
ნახ. 2.2. New Project ფანჯარა.



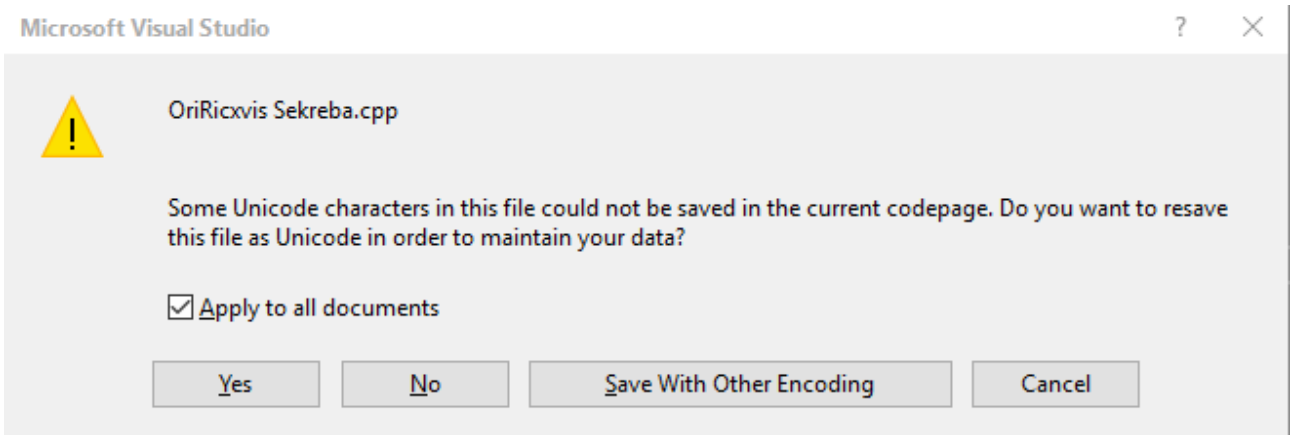
ნახ. 2.3. Win32 Application Wizard ფანჯარა.



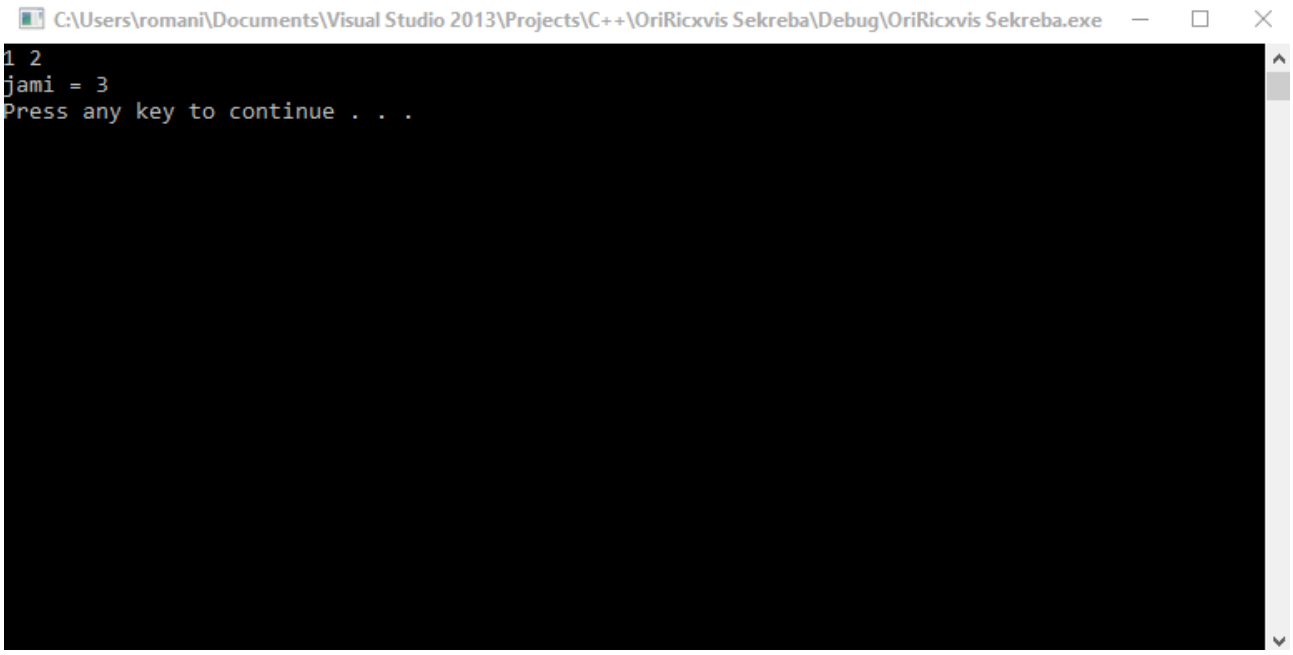
ნახ. 2.4. OriRicxvisShekreba.cpp გვერდი.



ნახ. 2.5. OriRicxvisShekreba.cpp გვერდი.



ნახ. 2.6. ფაილის შენახვა Unicode ფორმატში.



ნახ. 2.7. შედეგების ნახვა.

კომენტარები

C++ ენაში არსებობს ერთსტრიქონიანი და მრავალსტრიქონიანი კომენტარები. ერთსტრიქონიანი კომენტარი // სიმბოლოებით იწყება. მრავალსტრიქონიანი კომენტარი /* სიმბოლოებით იწყება და */ სიმბოლოებით მთავრდება. როგორც წესი, კომენტარი 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.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
/*
ეს არის ჩვენი                                                    ეს არის მრავალსტრიქონიანი კომენტარი
პირველი პროგრამა
*/
int ricxvi1, ricxvi2, jami;
//          რიცხვების შეტანა cin ფუნქციის გამოყენებით       ეს არის ერთსტრიქონიანი კომენტარი
cin >> ricxvi1 >> ricxvi2;
jami = ricxvi1 + ricxvi2;
cout << "jami = " << jami << endl;

system("pause");
return 0;
}
```

C++ ენის საბაზო ტიპები

ინფორმაციის სახესხვაობას, რომელსაც შეძლება ცვლადი შეიცავდეს, ***მონაცემის ტიპი*** ეწოდება. პროგრამის მონაცემები, ცვლადები და მუდმივები მონაცემთა გარკვეულ ტიპს უნდა ეკუთვნოდეს. ISO/ANSI C++ სტანდარტში განსაზღვრულია ***მონაცემების საბაზო ტიპები.*** ისინი სამ ნაწილად იყოფა:

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

ISO/ANSI C++ ტიპები მოცემულია 2.1 ცხრილში.

მთელრიცხვა ტიპები

მთელრიცხვა ცვლადები მხოლოდ მთელ რიცხვებს ინახავენ. C++ ISO/ANSI მთელრიცხვა ტიპები მოცემულია 2.1 ცხრილში. 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 (ორობითი თანრიგი) გამოიყენება. თუ ეს ბიტი ნულის ტოლია, მაშინ რიცხვი დადებითია, თუ ერთის ტოლი, მაშინ - უარყოფითი. უნიშნო რიცხვებში ნიშნის წარმოსადგენად უფროსი ბიტი არ გამოიყენება და შედის რიცხვის შემადგენლობაში. ამიტომ, უნიშნო რიცხვების აბსოლუტური მნიშვნელობა ორჯერ აღემატება ნიშნიანი რიცხვის მნიშვნელობას.

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;
```

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

მოძრავწერტილიანი ტიპები

მოძრავწერტილიანი ცვლადები წილად რიცხვებს ინახავენ. ქვემოთ მოცემულ პროგრამაში ხდება წილად რიცხვებთან მუშაობის დემონსტრირება.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამაში ხდება წილადებთან მუშაობის დემონსტრირება
float    wiladi1, wiladi2, shedegi1;
double wiladi3, wiladi4, shedegi2;

wiladi1 = 5.5;
wiladi3 = 7.98;
cin >> wiladi2 >> wiladi4;
shedegi1 = wiladi1 + wiladi2;
shedegi2 = wiladi3 + wiladi4;
cout << "shedegi1 = " << shedegi1 << "   shedegi2 = " << shedegi2 << endl;

system("pause");
return 0;
}
```

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

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

სიმბოლური ტიპები

სიმბოლური ცვლადი ერთ სიმბოლოს ინახავს. მას char ტიპი აქვს. char ტიპის ცვლადი ერთ ბაიტს იკავებს. მაგალითი.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამაში ხდება სიმბოლოებთან მუშაობის დემონსტრირება
char simbolo1 = 'w';
char simbolo2;
cin >> simbolo2;

cout << "simbolo1 = " << simbolo1 << "   simbolo2 = " << simbolo2 << endl;

system("pause");
```

```
return 0;
}
```

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,

```
cout << "Saba \n Ana" << endl;
```

აქ, cout ფუნქციის შესრულების შედეგად ფანჯრის პირველ სტრიქონში გამოჩნდება "Saba", მეორე სტრიქონში კი - "Ana". 2.2 ცხრილში მოცემულია მმართველი სიმბოლოები.

ცხრილი 2.2. 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

მმართველი სიმბოლოების გამოყენების შედეგები კარგად ჩანს კონსოლურ პროგრამებთან მუშაობისას. **კონსოლური პროგრამა** ჩვენთან ურთიერთქმედებს კლავიატურისა და ეკრანის საშუალებით, რომლებიც სიმბოლურ რეჟიმში მუშაობენ.

bool ტიპი

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
// პროგრამაში ხდება ლოგიკურ ცვლადთან მუშაობის დემონსტრირება
bool b1, b2;

b1 = true;
cin >> b2;
cout << "b1 = " << b1 << "   b2 = " << b2 << endl;

system("pause");
return 0;
```

```
}
```

true მნიშვნელობის მაგივრად უნდა შევიტანოთ 1, false მნიშვნელობის მაგივრად კი - 0.

string ტიპი

სტრიქონებთან სამუშაოდ შეგვიძლია გამოვიყენოთ string ტიპი. ამისათვის, პროგრამის ფაილს დასაწყისში უნდა დაემატოს #include "string" დირექტივა. მაგალითი:

```
#include "stdafx.h"
#include "iostream"
#include "string"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამაში ხდება სტრიქონებთან მუშაობის დემონსტრირება
string striqoni1 = "Ana da Saba";
string striqoni2;
cin >> striqoni2;
cout << striqoni1 << endl;
striqoni1 = striqoni2;
cout << striqoni1 << endl;

system("pause");
return 0;
}
```

სამწუხაროდ, cin ფუნქციის გამოყენებით შესაძლებელია მხოლოდ ისეთი სტრიქონების შეტანა, რომლებიც ინტერვალებს არ შეიცავს. ეს ფუნქცია ინტერვალს აღიქვამს სტრიქონის დასასრულად. იმისათვის, რომ შევძლოთ ინტერვალების შემცველი სტრიქონების შეტანა, უნდა გამოვიყენოთ getline() ფუნქცია. მოყვანილი პროგრამით ხდება ამ ფუნქციასთან მუშაობის დემონსტრირება.

```
#include "stdafx.h"
#include "iostream"
#include "string"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამაში ხდება ინტერვალების შემცველ სტრიქონებთან მუშაობის დემონსტრირება
string str1;
getline(cin, str1);                 //      შესატანი სტრიქონი: რომან სამხარაძე
cout << str1 << endl;

system("pause");
return 0;
}
```

როგორც ვხედავთ, ამ ფუნქციას ორი პარამეტრი აქვს. პირველია cin ფუნქცია, მეორე კი - სტრიქონული ცვლადი (str1), რომელშიც ჩაიწერება შეტანილი სტრიქონი.

ლიტერალები

ნებისმიერი სახის ფიქსირებულ მნიშვნელობას ***ლიტერალი*** ეწოდება. ლიტერალი (2.3 ცხრილი) შეიძლება იყოს მთელი რიცხვი, წილადი, სტრიქონი, სიმბოლო ან ლოგიკური მნიშვნელობა. მაგალითი:

```
#include "stdafx.h"
#include "iostream"
#include "string"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
int mteli1 = -4;                                    //    ლიტერალია - -4
long mteli2 = 45L;                                  //    ლიტერალია - 45L
double wiladi = 9.21;                               //    ლიტერალია - 9.21
bool logikuri = true;                               //    ლიტერალია - true
char simbolo = 'R';                                 //    ლიტერალია - 'R'
string striqoni = "Saba Samkharadze";               //    ლიტერალია - "Saba Samkharadze
cout << mteli1  << "   " << mteli2   << "   "
     <<  wiladi  << "   " << logikuri << "   "
     << simbolo << "   " << striqoni << endl;

system("pause");
return 0;
}
```

ცხრილი 2.3. სხვადასხვა ტიპის ლიტერალი.

ტიპი	მაგალითები
char, signed char, unsigned char	'A', 'S', '#', '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

კიდევ ერთი მაგალითი.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
```

```
	char c1 = 'a';
	signed char sc1 = '5';
	unsigned char uc1 = '2';
	unsigned int ui1 = 1;
	long l1 = 6;
	unsigned long ul1 = 7;
	float f1 = 4.4;
	double d1 = 5.5;
	long double ld1 = 2.2;
	bool b1 = true;
	cout << "c1 = " << c1 << "  sc1 = " << sc1 << "  uc1 = " << uc1 << endl
		<< "ui1 = " << ui1 << "  l1 = " << l1 << "  ul1 = " << ul1 << endl
		<< "f1 = " << f1 << "  d1 = " << d1 << "  ld1 = " << ld1 << endl
		 << "b1 = " << b1 << endl;

	system("pause");
	return 0;
}
```

მონაცემთა ტიპისთვის სინონიმის განსაზღვრა

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

typedef ***ტიპი სინონიმი***

საკვანძო სიტყვა typedef შეგვიძლია მივუთითოთ "int _tmain(int argc, _TCHAR* argv[])" სტრიქონის წინ:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

typedef unsigned int Ricxvebi;
typedef  long long Didi_Mteli;
int _tmain(int argc, _TCHAR* argv[])
{
Ricxvebi     mteli1 = 9;
Didi_Mteli mteli2 = 795LL;
cout << "mteli1 = " << mteli1 << "   mteli2 = " << mteli2 << endl;

system("pause");
return 0;
}
```

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 აადვილებს 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; წილადი რიცხვები: 23.54, 1.09; სიმბოლოები: 'Q', 'g'; სტრიქონი: "Visual C++" და ა.შ. მოცემული პროგრამით ხდება მუდმივასთან მუშაობის დემონსტრირება.

```
#include "stdafx.h"
#include "iostream"
#include "string"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამით ხდება მუდმივასთან მუშაობის დემონსტრირება
//      მუდმივას განსაზღვრა
const int mteli_mudmiva = 1024;
const char simbolo_mudmiva = 'a';
const float wiladi_mudmiva1 =  12.09;
const double wiladi_mudmiva2 = 456.302;
const string str1 = "C++";
int cvladi, jami;
double shedegi;

cin >> cvladi;
jami = cvladi + mteli_mudmiva;
shedegi = wiladi_mudmiva1 + wiladi_mudmiva2;
cout << jami << endl;
cout << simbolo_mudmiva << endl;
cout << wiladi_mudmiva1 << endl;
cout << shedegi << endl;
cout << str1 << endl;

system("pause");
return 0;
}
```

ცვლადი და მისი ინიციალიზება

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

ტიპი ცვლადის_სახელი;

სადაც, ***ტიპი*** ცვლადის ტიპია, ***ცვლადის_სახელი*** - კი მისი სახელი. ნებისმიერი ცვლადი უნდა

გამოცხადდეს მის გამოყენებამდე. წინააღმდეგ შემთხვევაში, აღიძვრება შეცდომა. 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 მის გამოყენებამდე უნდა მივანიჭოთ. ამის გაკეთება შეიძლება ცვლადის გამოცხადებისას ან გამოცხადების შემდეგ. ამ პროგრამით ხდება ცვლადების ინიციალიზების დემონსტრირება.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამით ხდება ცვლადების ინიციალიზების დემონსტრირება
int ricxvi3, ricxvi1 = 25, ricxvi2 = 30;
char simbolo = L'S';
double wiladi1 = 150.75;
cout << ricxvi1 << "   " << wiladi1 << "   " << simbolo << endl;

system("pause");
return 0;
}
```

ცვლადების დინამიკური ინიციალიზება

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამით ხდება ცვლადის დინამიკური ინიციალიზების დემონსტრირება
int simagle = 5, sigane, sigrdze;

cin >> sigane  >> sigrdze;
// moculoba ცვლადის ინიციალიზება დინამიკურად  ხდება
int moculoba = simagle * sigane * sigrdze;
cout << moculoba;
```

```
system("pause");
return 0;
}
```

ოპერატორები

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

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.

არითმეტიკის ოპერატორები

არითმეტიკის ოპერატორები (2.4 ცხრილი) შეგვიძლია გამოვიყენოთ როგორც მთელი, ისე წილადი რიცხვისთვის.

მოვიყვანოთ ზოგიერთი განმარტება. როდესაც გაყოფის ოპერატორს ვიყენებთ მთელი რიცხვებისთვის, მაშინ შედეგი იქნება მთელი რიცხვი, ნაშთი კი უარიყოფა. მაგალითად, 20 / 6 გაყოფის შედეგი იქნება მთელი რიცხვი 3. იმისათვის, რომ განაყოფი იყოს წილადი საჭიროა, რომ მრიცხველი ან მნიშვნელი ან ორივე ერთად იყოს წილადი. 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 2.4. 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.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამით ხდება არითმეტიკის ოპერაციების მუშაობის
//      დემონსტრირება მთელი რიცხვებისათვის
int ricxvi1, ricxvi2, jami, sxvaoba, namravli, ganayofi, nashti;

cin >> ricxvi1 >> ricxvi2;
jami       = ricxvi1 + ricxvi2;
sxvaoba  = ricxvi1 - ricxvi2;
namravli = ricxvi1 * ricxvi2;
ganayofi = ricxvi1 / ricxvi2;
nashti     = ricxvi1 % ricxvi2;
cout << "jami = " << jami << "   sxvaoba = " << sxvaoba << endl;
cout << "namravli = " << namravli << endl;
cout << "ganayofi = " << ganayofi << "   nashti = " << nashti << endl

system("pause");
```

```
return 0;
}
```

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      პროგრამით ხდება არითმეტიკის ოპერატორების მუშაობის
//      დემონსტრირება წილადი რიცხვებისათვის
double wiladi1, wiladi2, jami, sxvaoba, namravli, ganayofi, nashti;
int ricxvi1, ricxvi2;

cin >> ricxvi1 >> ricxvi2;
cin >> wiladi1 >> wiladi2;
jami       = wiladi1 + wiladi2;
sxvaoba  = wiladi1 - wiladi2;
namravli = wiladi1 * wiladi2;
ganayofi = wiladi1 / wiladi2;
nashti   = ricxvi1 % ricxvi2;
cout << "jami = " << jami << "   sxvaoba = " << sxvaoba << endl;
cout << "namravli = " << namravli << endl;
cout << "ganayofi = " << ganayofi << "  nashti = " << nashti << endl;

system("pause");
return 0;
}
```

ოპერატორების პრიორიტეტი

ჩვეულებრივ, გამოსახულებაში ჯერ სრულდება პრიორიტეტული ოპერაციები, შემდეგ კი - ნაკლებად პრიორიტეტული. 2.5 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 ოპერაციები შემდეგი მიმდევრობით შესრულდება: 1) პირველ მრგვალ ფრჩხილებში მოთავსებული გამოსახულება; 2) * ოპერატორი; 3) მეორე მრგვალ ფრჩხილებში მოთავსებული გამოსახულება; 4) / ოპერატორი; 5) - ოპერატორი.

როგორც ვხედავთ, პრიორიტეტების ცოდნა საჭიროა იმისთვის, 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.

ცხრილი 2.5. 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

ინკრემენტისა და დეკრემენტის ოპერატორები

ინკრემენტის ოპერატორი (++) ერთით ზრდის თავისი ოპერანდის (არგუმენტის)

მნიშვნელობას. დეკრემენტის ოპერატორი (--) ერთით ამცირებს თავისი ოპერანდის მნიშვნელობას. ოპერანდი არის ცვლადი, რომელზეც სრულდება ინკრემენტის ან დეკრემენტის ოპერაცია. განვიხილოთ მაგალითები.

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.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      ++ ოპერატორის მუშაობის დემონსტრირება
int incr, shedegi;

incr = 5;
//      პრეფიქსური ინკრემენტი
shedegi = ++incr;                                       //      incr    = 6, shedegi = 6
cout << "incrementi = " << incr << "   shedegi = " << shedegi << endl;
incr = 5;
//      პოსტფიქსური ინკრემენტი
shedegi = incr++;                                       //      shedegi = 5, incr    = 6
cout << "incrementi = " << incr << "   shedegi = " << shedegi << endl;

system("pause");
return 0;
}
```

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.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      -- ოპერატორის მუშაობის დემონსტრირება
int decr, shedegi;

decr = 5;
//      პრეფიქსური დეკრემენტი
shedegi = --decr;                                       //      decr    = 4, shedegi = 4
cout << "decrementi = " << decr << "shedegi = " << shedegi << endl;
decr = 5;
//      პოსტფიქსური დეკრემენტი
shedegi = decr--;                                       //      shedegi = 5, decr    = 4
cout << "decrementi = " << decr << "shedegi = " << shedegi<< endl;

system("pause");
return 0;
}
```

შედარებისა და ლოგიკის ოპერატორები

შედარების ოპერატორი (2.6 ცხრილი) ადარებს ორ მნიშვნელობას და გასცემს bool ტიპის true ან false მნიშვნელობას. ლოგიკის ოპერატორი (2.7 ცხრილი) ასრულებს ლოგიკის ოპერაციას bool ტიპის მნიშვნელობებზე.

C++ ენაში == და != ოპერატორები შეიძლება გამოვიყენოთ ყველა ობიექტის მიმართ მათი შედარებისთვის 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.

2.8 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 2.6. შედარების ოპერატორები

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

ცხრილი 2.7. 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 2.8. 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.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//          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
cout << b3 << endl;
cout << b4 << endl;
cout << b5 << endl;
cout << b6 << endl;
```

```
system("pause");
return 0;
}
```

მოცემული პროგრამით ხდება შედარების ოპერატორების მუშაობის დემონსტრირება.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      შედარების ოპერატორების მუშაობის დემონსტრირება
int ricxvi1 = 5, ricxvi2 = 10;
bool shedegi;

shedegi = ricxvi1 > 0;                          //      shedegi = true
cout << shedegi << endl;
shedegi = ( ricxvi1 < 2 ) && ( ricxvi2 > 7 );   //      shedegi = false
cout << shedegi << endl;
shedegi = ( ricxvi1 == 3 ) || ( ricxvi2 != 5 ); //      shedegi = true
cout << shedegi << endl;

system("pause");
return 0;
}
```

ამ პროგრამის „shedegi = ricxvi1 > 0;“ სტრიქონში ricxvi1 > 0 გამოსახულება true მნიშვნელობას გასცემს, რადგან ricxvi1 = 5. true მნიშვნელობა shedegi ცვლადს მიენიჭება. „shedegi = (ricxvi1 < 2) && (ricxvi2 > 7);“ სტრიქონში ricxvi1 < 2 გამოსახულება false მნიშვნელობას გასცემს, რადგან ricxvi1 = 5, ხოლო ricxvi2 > 7 გამოსახულება კი - true მნიშვნელობას, რადგან ricxvi2 = 10. მივიღებთ false && true გამოსახულებას, რომელიც false მნიშვნელობას გასცემს. ეს მნიშვნელობა shedegi ცვლადს მიენიჭება. ანალოგიურად გამოითვლება „shedegi = (ricxvi1 == 3) || (ricxvi2 != 5);“ გამოსახულების მნიშვნელობა.

&& 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.

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 ოპერაციების 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 (2.9 ცხრილი). თუმცა, #include "math.h" დირექტივის მითითება აუცილებელი არ არის. მათემატიკის ფუნქციების უმრავლესობა გასცემს double ტიპის შედეგს და აქვს double ტიპის პარამეტრი.

ცხრილი 2.9. 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
pow(x, y)	ახარისხება
floor(x)	დამრგვალება ნაკლებობით
ceil(x)	დამრგვალება მეტობით

შევადგინოთ პროგრამა, რომლის შეასრულებს კვადრატული ფესვის ამოღებას რიცხვიდან.

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//        კვადრატული ფესვის ამოღება
double ricxvi, shedegi;

cin >> ricxvi;
shedegi = sqrt(ricxvi);
cout << "shedegi = " << shedegi << endl;

system("pause");
return 0;
}
```

შევადგინოთ კიდევ ერთი პროგრამა, რომელიც გამოთვლის მოცემული გამოსახულების მნიშვნელობას:

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//          ფორმულის გამოთვლა
double ricxvi, shedegi;

cin >> ricxvi;
shedegi =  sqrt(pow(ricxvi,5) + sin(ricxvi)) / ( 1 - exp(ricxvi) );
cout << shedegi << endl;

system("pause");
return 0;
}
```

დანართი 1-ში 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 ოპერაციებისგან 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:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
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;
cout << wiladi_d << endl;

system("pause");
return 0;
}
```

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 ტიპისგან. ამან შეიძლება გამოიწვიოს მნიშვნელობის შეცვლა და შესაბამისად, მონაცემების დაკარგვა. მაგალითი:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//          პროგრამით ხდება არაცხადი გარდაქმნის დემონსტრირება
int mteli = 0;
float wiladi = 3.7f;
mteli = wiladi;                                    //          mteli ცვლადს მიენიჭება 3
cout << mteli << endl;

system("pause");
return 0;
}
```

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:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//          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
cout << mteli << endl;

system("pause");
return 0;
}
```

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 25, რომელიც მიენიჭება mteli ცვლადს.

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//          ინფორმაციის დაკარგვის დემონსტრირება
double wiladi1 = 0.8;
float    wiladi2 = 0.7f;
long mteli1;
int   mteli2;

mteli1 = wiladi1;                              //          mteli1 = 0
mteli2 = wiladi2;                              //          mteli2 = 0
cout << "mteli1 = " << mteli1 << "   mteli2 = " << mteli2 << endl;

system("pause");
```

```
return 0;
}
```

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.

ტიპების დაყვანის ძველი სტილი

ტიპების დაყვანის ძველი სტილის სინტაქსია:

(დაყვანის_ტიპი) გამოსახულება;

მაგალითი:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
// ტიპების დაყვანის ძველი სტილის დემონსტრირება
double wiladi1 = 45.8;
double wiladi2 = 20.3;
int mteli;
mteli = (int) wiladi1 - (int) wiladi2;                    //        mteli = 25
cout << mteli << endl;

system("pause");
return 0;
}
```

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

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      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
cout << "shida ricxvi1 = " << ricxvi1 << "\nshida ricxvi3 = " << ricxvi3 << endl;
//      ხილვადობის ახალი უბნის დასასრული
}
ricxvi1 += 10;                  //      გარე ricxvi1 = 20
ricxvi2 += 20;                  //      გარე ricxvi2 = 40
cout << "gare ricxvi1 = " << ricxvi1 << "\ngare ricxvi2 =  " << ricxvi2 << endl;
```

```
system("pause");
return 0;
}
```

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 აქვს. 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:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
//      სტატიკურ ცვლადთან მუშობის დემონსტრირება
static int ricxvi1 = 5;
{
static int ricxvi2 = 10;
ricxvi2++;
cout << ricxvi2 << endl;
}
ricxvi1++;
cout << ricxvi1 << endl;

system("pause");
return 0;
}
```

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. მას 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, ის არსებობს პროგრამის დაწყების მომენტიდან მისი დამთავრების მომენტამდე. თუ გლობალურ ცვლადს არ აქვს მინიჭებული საწყისი მნიშვნელობა, მაშინ მისი ნაგულისხმევი მნიშვნელობაა ნული. გლობალური ცვლადის გამოცხადება შეგვიძლია int _tmain(int argc, _TCHAR* argv[]) სტრიქონის წინ. მაგალითი:

```
//          გლობალურ ცვლადებთან მუშობის დემონსტრირება
#include "stdafx.h"
#include "iostream"
using namespace std;

int ricxvi1 = 10;
static int ricxvi2 = 20;
int _tmain(int argc, _TCHAR* argv[])
{
int ricxvi3, ricxvi4, shedegi;

ricxvi3 = 30;
ricxvi4 = 40;
shedegi = ricxvi1 + ricxvi2 + ricxvi3 + ricxvi4;          //          shedegi = 100
cout << shedegi << endl;

system("pause");
return 0;
}
```

აქ ricxvi1 და ricxvi2 გლობალური ცვლადებია. 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:

```
#include "stdafx.h"
#include "iostream"
using namespace std;

int ricxvi1 = 10;
static int ricxvi2 = 20;
int _tmain(int argc, _TCHAR* argv[])
{
int ricxvi1 = 88;          //          ლოკალური ricxvi1 ცვლადი ფარავს გლობალურ ricxvi1 ცვლადს
int ricxvi2 = 77;          //          ლოკალური ricxvi2 ცვლადი ფარავს გლობალურ ricxvi2 ცვლადს
int ricxvi3 = ricxvi1 + ricxvi2;
cout << ricxvi3 << endl;                              //          გაიცემა 165
system("pause");
return 0;
}
```