

```
#include <iostream>
using namespace std;
class Rational {
public:
    Rational(long n=0, long d=1)
    {
        nmr = n; dnm = d;
        this->normalize();
    }
    friend Rational operator + (const Rational &x, const Rational &y);
    friend Rational operator - (const Rational &x, const Rational &y);
    friend Rational operator * (const Rational &x, const Rational &y);
    friend Rational operator / (const Rational &x, const Rational &y);
    friend void operator += (Rational &x, const Rational &y);
    friend void operator -= (Rational &x, const Rational &y);
    friend void operator *= (Rational &x, const Rational &y);
    friend void operator /= (Rational &x, const Rational &y);
    friend bool operator == (const Rational &x, const Rational &y);
    friend bool operator < (const Rational &x, const Rational &y);
    friend bool operator > (const Rational &x, const Rational &y);
    void show() const;
};
```

C++ -ի

Հիմունքները

```
Rational operator + (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm + y.nmr * x.dnm, x.dnm * y.dnm);
}
```

Հեղինակներ՝

```
Rational operator - (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm - y.nmr * x.dnm, x.dnm * y.dnm);
}
```

Ռուբեն Վարդանյան Սուրեն Կարապետյան

```
Rational operator * (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.nmr, x.dnm * y.dnm);
}
```

```
Rational operator / (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm, x.dnm * y.nmr);
}
```

```
void operator += (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.dnm + y.nmr * x.dnm;
    x.dnm = x.dnm * y.dnm;
    x.normalize();
}
```

```
void operator -= (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.dnm - y.nmr * x.dnm;
    x.dnm = x.dnm * y.dnm;
    x.normalize();
}
```

```
void operator *= (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.nmr;
    x.dnm = x.dnm * y.dnm;
    x.normalize();
}
```

Վեբ կայք՝ www.pocpp.org

© 2007 Ռ.Չ. Վարդանյան և Ս.Գ. Կարապետյան

Բովանդակություն

Ներածություն.....	6
Թարգմանիչներ	6
Console ծրագրերի նախագծեր Microsoft Visual C++ 6 - նվ	6
Նոր նախագծի ստեղծում	6
Ծրագրի կոմպիլացիա (Compilation) և աշխատեցում (Execution)	9
Console ծրագրերի նախագծեր Microsoft Visual C++ 2005 - նվ	9
Նոր նախագծի ստեղծում	9
Ծրագրի կոմպիլացիա (Compilation) և աշխատեցում (Execution)	13
Բաժին 1.1 C++ - ի կառուցվածքը	15
Մեկնաբանություններ (Comments)	18
Բաժին 1.2 Փոփոխականներ: Տվյալների տիպեր: Հաստատուններ.....	19
Նույնարկիչներ (Identifiers).....	19
Տվյալների տիպեր (Data Types).....	20
Փոփոխականների հայտարարումը (Declaration of Variables).....	22
Փոփոխականների սկզբնարժեքավորում (Initialization of Variables)	23
Փոփոխականների տեսանելիության տիրույթ (Scope of Variables)	24
Հաստատուններ (Constants)	24
Ամբողջ թվեր (Integer Numbers).....	24
Լողացող կետով թվեր (Floating Point Numbers).....	25
Սիմվոլներ և տողեր (Characters and strings)	25
Նշանակված հաստատուններ (Defined Constants) (#define)	26
Հայտարարված հաստատուններ (Declared constants) (const)	27
Բաժին 1.3 Օպերատորներ	28
Վերագրում ` =	28
Թվաբանական գործողություններ` +, -, *, /, %	29
Բարդ վերագրման օպերատորներ` +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =	29
Մեծացման և փոքրացման օպերատորներ` ++, --	30
Համեմատության օպերատորներ` ==, !=, >, <, >=, <=	31
Տրամաբանական օպերատորներ` !, &&, 	31
Պայմանական օպերատոր` ?	32
Բիթային օպերատորներ` &, , ^, ~, <<, >>	32
Տիպերի ձևափոխման օպերատորներ.....	33
sizeof() օպերատորը	33
Օպերատորների նախապատվությունը	34
Բաժին 1.4 Հաղորդակցություն օգտագործողի հետ.....	36
Ելք (cout).....	36
Մուտք (cin)	37
Բաժին 2.1 Կառավարման համակարգեր (Control Structures)	38

Պայմանայան համակարգ if և else.....	39
Կրկնվող համակարգեր կամ ցիկլեր (Repetitive Structures or Loops)	40
while ցիկլը.....	40
do - while ցիկլը	41
for ցիկլը.....	42
Անցման հրամաններ (Jumps)	43
break հրամանը.....	43
continue հրամանը.....	44
goto հրամանը.....	44
Ընտրության ստրուկտուրա` switch.....	45
Բաժին 2.2 Ֆունկցիաներ (I)	47
Ֆունկցիաներ` առանց տիպի: void - ի օգտագործումը.....	50
Բաժին 2.3 Ֆունկցիաներ (II)	52
Արգումենտների փոխանցումը արժեքով (by value) և հասցեով (by reference)	52
Արգումենտների լռության (Default) արժեքներ	54
Ֆունկցիաների գերբեռնում (Overloading)	55
inline (տողամիջյան) ֆունկցիաներ	56
Բեկուրսիա (Recursivity).....	56
Ֆունկցիայի նախատիպ (Prototype).....	57
Բաժին 3.1 Չանգվածներ (Arrays).....	59
Չանգվածների սկզբնարժեքավորումը (Initialization of Arrays).....	59
Չանգվածների տարրերին դիմումը (Access to the Values of an Array)	60
Բազմաչափ զանգվածներ (Multidimensional Arrays)	62
Չանգվածները` որպես պարամետրեր	63
Բաժին 3.2 Միավորային տողեր (Strings of Characters).....	66
Տողերի սկզբնարժեքավորում	66
Արժեքների վերագրումը տողերին.....	67
Տողերի փոխակերպումն այլ տիպերի	71
Բաժին 3.3 Ցուցիչներ (Pointers)	72
Հասցեավորման (Address) օպերատոր (&).....	72
Հետհասցեավորման (Reference) օպերատոր (*)	73
«Ցուցիչ» տիպի փոփոխականների հայտարարումը	74
const –ի օգտագործումը ցուցիչներում	75
Ցուցիչներ և զանգվածներ	76
Ցուցիչների սկզբնարժեքավորում	77
Ցուցիչների թվաբանությունը.....	78
Ցուցիչ`	80
void ցուցիչներ	80
Ցուցիչ ֆունկցիային	81
Բաժին 3.4 Դինամիկ հիշողություն (Dynamic Memory).....	83
new և new [] օպերատորները.....	83
delete օպերատորը	84
Բաժին 3.5 Կառուցվածքներ (Structures)	86

Ցուցիչներ կառուցվածքներին.....	89
Կառուցվածքների խտացում	91
Բաժին 3.6 Օգտագործողի հայտարարած տիպեր	92
Սեփական տիպերի հայտարարում (typedef)	92
Միավորումներ (Unions)	92
Անանուն (Anonymous) միավորումներ	93
Թվարկումներ (Enumerations) (enum)	94
Բաժին 4.1 Դասեր (Classes)	96
Կառուցիչներ և փլուզիչներ (Constructors and Destructors).....	100
Կառուցիչների գերբեռնում (Overloading Constructors).....	102
Ցուցիչներ դասերին	103
struct բանալի-բառով որոշված դասեր	104
Բաժին 4.2 Օպերատորների գերբեռնում (Overloading Operators)	105
this բանալի-բառը.....	108
Ունար օպերատորների ծանրաբեռնում	109
Ստատիկ անդամներ	111
Ցուցիչ դասի անդամների վրա.....	113
Վերագրման օպերատորի ծանրաբեռնում	114
Ֆունկցիայի կանչ օպերատորի ծանրաբեռնում	115
const բանալի-բառի այլ կիրառությունները.....	116
Դասերի օգտագործման օրինակ՝ ռացիոնալ թվերի դաս	119
Բաժին 4.3 Դասերի հարաբերությունները.....	124
Բարեկամ ֆունկցիաներ (Friend Functions)	124
Ռացիոնալ թվերի դաս	125
Բարեկամ դասեր (Friend Classes).....	127
Դասերի ժառանգականությունը (Inheritance Between Classes)	129
Ի՞նչ է ժառանգվում ժառանգվող դասից.....	131
Բազմակի ժառանգականություն (Multiple Inheritance)	133
Բաժին 4.4 Բազմաձևություն (Polymorphism).....	134
Ժառանգվող դասերի ցուցիչներ	134
Վիրտուալ անդամներ.....	135
Աբստրակտ դասեր.....	136
Բաժին 5.1 Կաղապարներ (Templates)	140
Ֆունկցիաների կաղապարներ.....	140
Դասերի	143
Կաղապարի պարամետրեր	144
Կաղապարներ	145
Բաժին 5.2 Նախաթարգմանչի հրամաններ (Preprocessor Directives)	146
#define.....	146
#undef	146
#ifdef, #ifndef, #if, #endif, #else and #elif	147
#line.....	148
#error.....	148
#include.....	148
#pragma.....	149

Բաժին 6.1 Մուտք/Ելք ֆայլերի հետ	150
Ֆայլի բացում.....	150
Ֆայլի փակում	151
Ֆայլերի հետ աշխատանք տեքստային ռեժիմում.....	151
Վիճակի դրոշակների ստուգում	153
get և put հոսքային ցուցիչներ	153
Երկուական ֆայլեր	154
Բուֆերներ և Սինխրոնիզացիա.....	155
Բաժին 7.1	156
using.....	157
Անվանատարածքների այլընտրանքային անուններ	159
std անվանատարածք	159

Ներածություն

Այս գիրքը նախատեսված է այն մարդկանց համար, ովքեր ցանկանում են սովորել ծրագրավորել **C++** միջավայրում: Ծրագրավորման մեկ այլ լեզվի իմացությունը, իհարկե, կհեշտացնի ուսումնական պրոցեսը, սակայն գիրքը կազմված է այնպես, որ կարող է օգտագործվել նաև ծրագրավորման ոչ մի լեզու չիմացող մարդկանց կողմից:

Գրքում ներկայացված են լեզվի հիմնական գաղափարները, տերմինների անգլերեն ներկայացումները, ինչպես նաև օրինակների փորձարկման համար անհրաժեշտ ծրագրերի օգտագործման եղանակները:

Թարգմանիչներ

Այս գրքում գրված բոլոր օրինակները **Console** ծրագրեր են, այսինքն՝ նրանք աշխատում են տեքստային միջավայրում, որը թույլ է տալիս նրանց կարդալ մուտքագրվող տվյալները և տպել արդյունքը:

C++ - ի բոլոր թարգմանիչները կարող են արտադրել **Console** ծրագրեր: Այս գրքում բերված են բացատրություններ, թե ինչպես կարելի է արտադրել ծրագրեր **Microsoft Visual C++ 6** - ի և **Microsoft Visual C++ 2005** - ի թարգմանիչով:

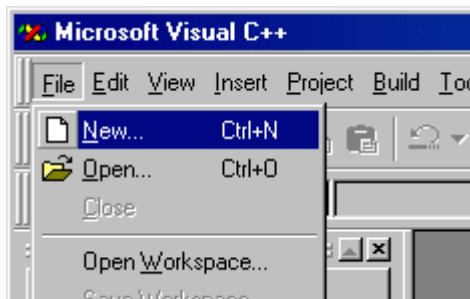
Console ծրագրերի նախագծեր Microsoft Visual C++ 6 - ու

Այս թարգմանիչը ինտեգրացված է **Microsoft Visual Studio 6** ծրագրավորման միջավայրում: Այս միջավայրում ծրագրեր ստեղծելու ամենահեշտ եղանակը նախագծեր ստեղծելն է: Այժմ կստեղծենք **test** անունով նախագիծ:

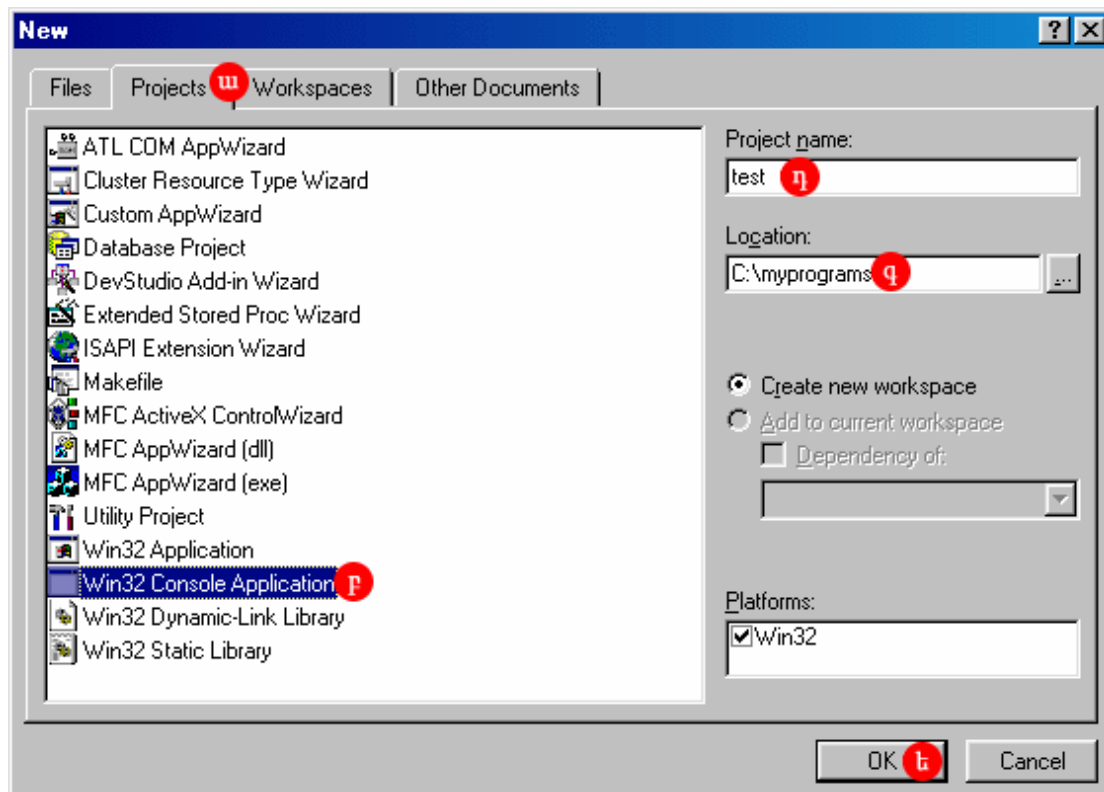
Նոր նախագծի ստեղծում

1. Աշխատեցնենք **Microsoft Visual C++ 6** միջավայրը:

Երբ հայտնվի պատուհանը, սեղմենք **File**, այնուհետև՝ **New**:



2. Կհայտնվի **New** անունով պատուհանը, որն ունի հետևյալ տեսքը.



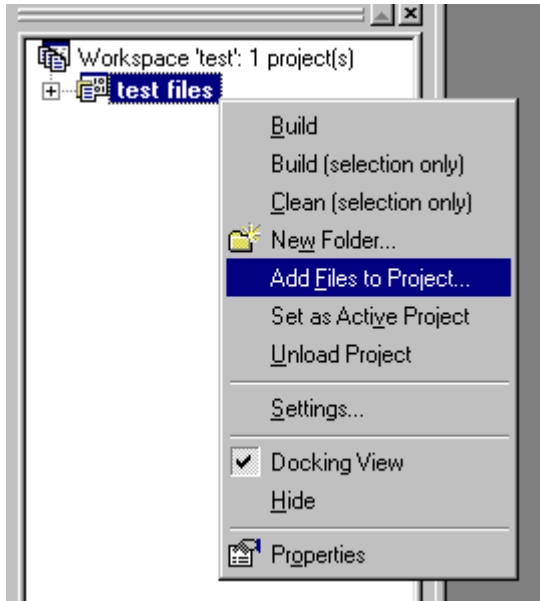
Կատարենք հետևյալ քայլերը.

- ա. ընտրենք **Projects** բաժինը,
- բ. ընտրենք նախագծի **Win32 Console Application** տիպը,
- գ. գրենք այն պանակի հասցեն, որտեղ ուզում ենք պահել ծրագիրը,
- դ. մեր նախագծին անուն տանք, օրինակ՝ **test**,
- ե. սեղմենք **OK** կոճակը:

Կհայտնվի մի պատուհան, որը կհարցնի, թե ինչպիսի նախագիծ ենք ցանկանում ստեղծել: Ընտրենք **An empty project** (դատարկ նախագիծ) և սեղմենք **Finish** կոճակը:

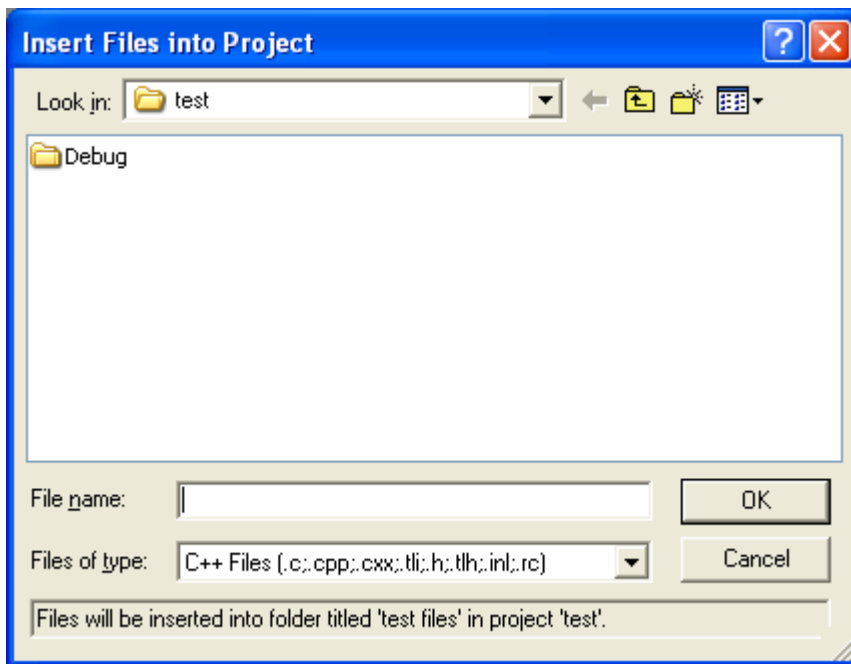
3. Այժմ ունենք դատարկ նախագիծ: Ծրագրի պատուհանի ներքևի ձախ մասում կտեսնենք երկու բաժիններ՝ **ClassView** և **FileView**: Ընտրենք **FileView** բաժինը:



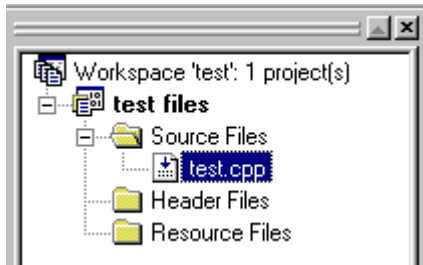


4. **FileView** բաժնում կա **test files** (փոխարինեք **test** բառը այն անունով, որը տվել եք նախագծին **2**դ քայլում) անունով մի խումբ: Պահելով մկնիկի ցուցիչը այդ խմբի վրա՝ սեղմենք աջ կոճակը և բերված ցուցակից ընտրենք **Add Files to Project...** :

5. Այս պատուհանի միջոցով կարելի է նախագծին ֆայլեր ավելացնել: Նախագծին կարող ենք ավելացնել կամ արդեն գոյություն ունեցող ֆայլ, կամ էլ ստեղծել նորը՝ գրելով նոր ֆայլի անունը **File name** տեքստային պատուհանի մեջ: Ստեղծված ֆայլի վերջավորությունը կլինի **cpp**, ինչը նշանակում է **C Plus Plus**: Նոր ֆայլ ստեղծելիս ծրագիրը կհարցնի՝ արդյոք ցանկանում ենք ստեղծել նոր ֆայլ: Պատասխանենք՝ այո (**Yes**):




6. Նախագծին ֆայլեր ավելասնելիս՝ դրանք կհայտնվեն **test files** խմբի տակ՝ **Source Files** պանակում: Ֆայլի վրա մկնիկի ձախ կոճակի կրկնակի սեղմումով՝ ֆայլի պարունակությունը կբացվի նոր բացված պատուհանի մեջ, և կկարողանանք աշխատել դրա հետ:



Ծրագրի կոմպիլացիա (Compilation) և աշխատեցում (Execution)

Եթե ծրագիրն արդեն պատրաստ է, և ուզում ենք այն աշխատեցնել, ապա կարող ենք մենյուից ընտրել **Build** ցանկը և բացված ցուցակից ընտրել **Execute test.exe** (**test.exe** բառը կփոխվի՝ կախված ընտրված նախագծի անունից) տողը:

Այս գործողությունը կարելի է կատարել նաև  սիմվոլով կոճակը սեղմելով:

Եթե ծրագրում սխալներ չկան, ապա այն աշխատացնելուց հետո կտեսնենք սպասվելիք արդյունքը, հակառակ դեպքում էկրանի ներքևի մասում կերևա հաղորդագրություն սխալի մասին (ֆայլի անունը, տողի համարը, սխալի բնույթը):

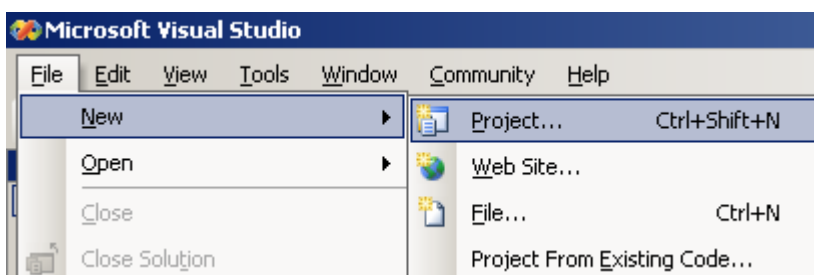
Console ծրագրերի նախագծեր Microsoft Visual C++ 2005 - ու

Այս թարգմանիչը ներդրված է **Microsoft Visual Studio 2005** ծրագրավորման միջավայրում: Ինչպես և նախկին **Visual Studio** - ի տարբերակներում, այս միջավայրում ծրագրեր ստեղծելու ամենահեշտ եղանակը նախագծեր ստեղծելն է: Այժմ կստեղծենք **test** անունով նախագիծ:

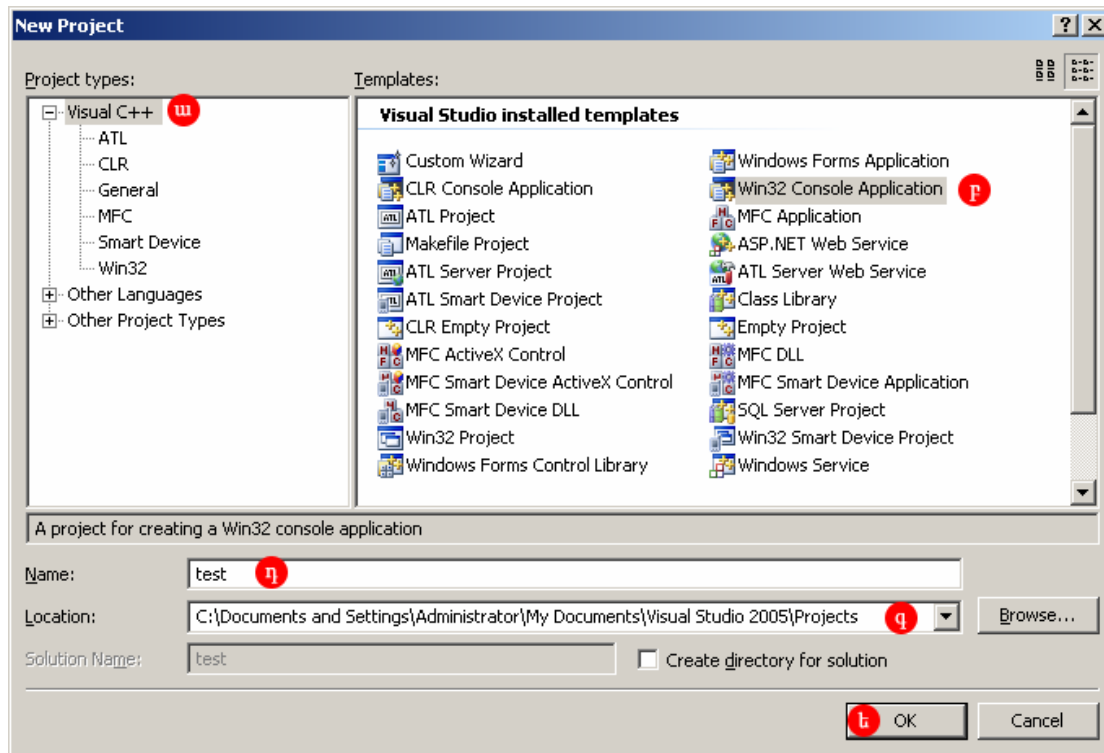
Նոր նախագծի ստեղծում

1. Աշխատեցնենք **Microsoft Visual Studio 2005** միջավայրը:

Երբ հայտնվի պատուհանը, ընտրենք **File > New > Project ...**:



2. Կհայտնվի **New Project** անունով պատուհանը, որն ունի հետևյալ տեսքը՝



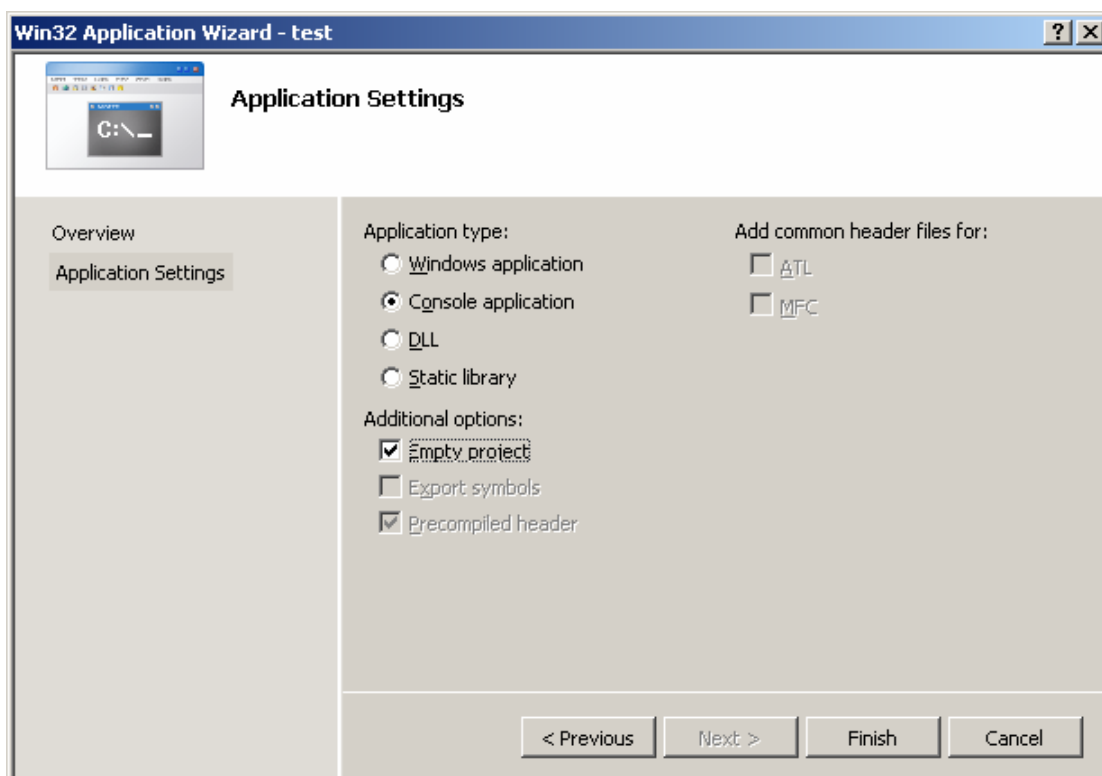
Կատարենք հետևյալ քայլերը.

- ա. ընտրենք **Visual C++** բաժինը
- բ. ընտրենք նախագծի **Win32 Console Application** տեսակը,
- գ. գրենք այն պանակի հասցեն, որտեղ ուզում եք պահել ծրագիրը,
- դ. նախագծին անուն տանք, օրինակ՝ **test**,
- ե. սեղմենք **OK** կոճակը:

Հայտնված պատուհանի միջոցով կարող ենք ստեղծել տրված տիպի մեր նախընտրած նախագիծը:

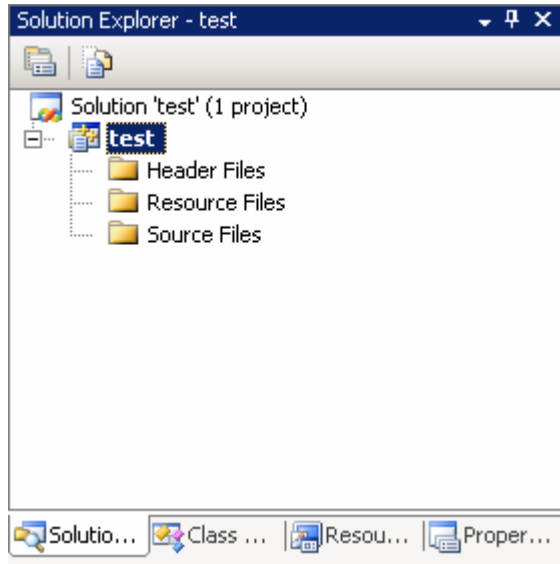


Պատուհանի ձախ մասում բերված են հնարավոր բաժինները, որոնք կարող ենք փոփոխել, և **Overview** էջը, որն ուղղակի ամփոփում է նախագիծը: Կախված նախագծի տեսակից, էջերի քանակը կարող է փոփոխվել: Մեր դեպքում՝ **Win32 Console Application** տիպի նախագծերում, կա միայն մեկ էջ՝ **Application Settings**, որը թույլ է տալիս փոփոխել նախագծի հիմնական պարամետրերը: Սեղմենք **Next** կոճակը, որպեսզի անցնենք այդ էջին:

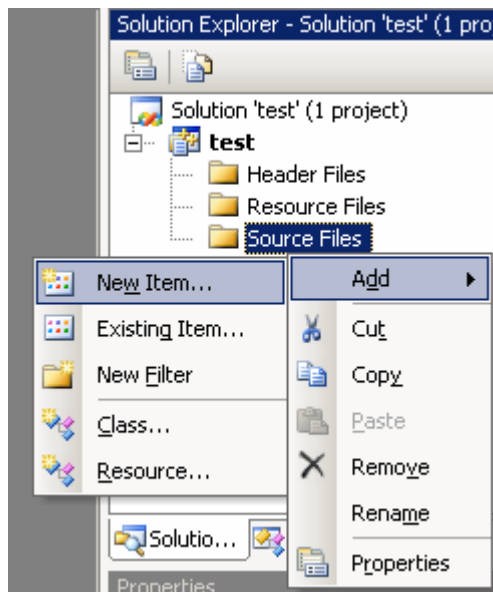


Այս էջում, որպես **Application type** ընտրենք **Console application** - ը ու **Additional options** - ից ընտրենք **Empty project** – ը: Սեղմեք **Finish** կոճակը:

3. Այժմ ունենք դատարկ նախագիծ: Աշխատանքի համար մեզ անհրաժեշտ կլինի **Solution Explorer** պատուհանը: Այն ունի հետևյալ տեսքը՝

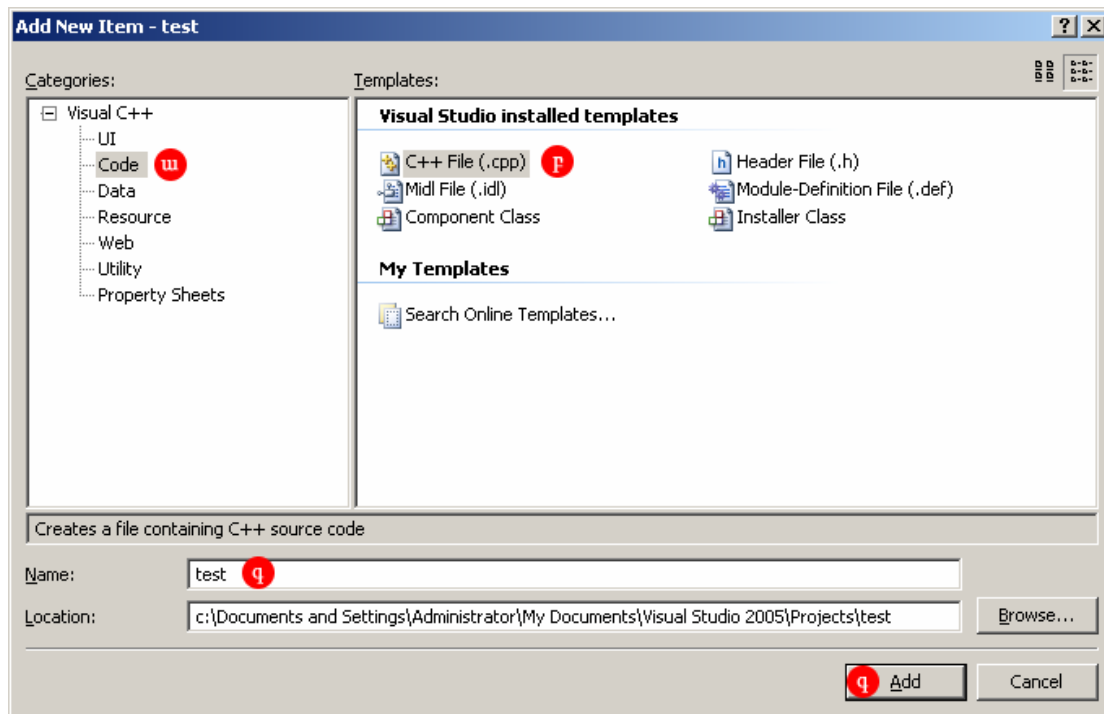


Եթե այն չի բացված, ապա ընտրենք **View > Solution Explorer** կամ պարզապես սեղմենք **Ctrl + Alt + L**:

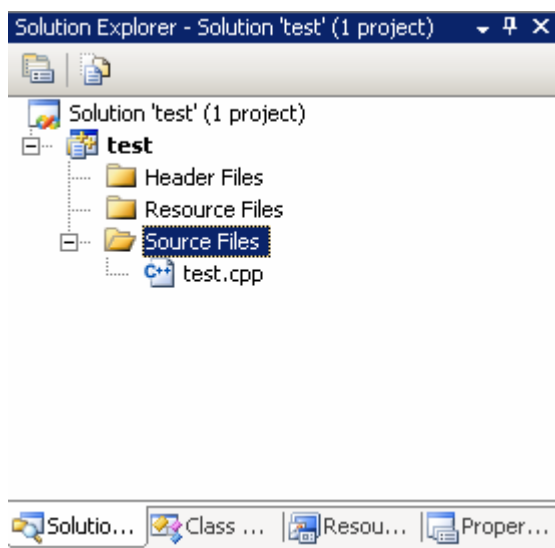


4. Հաջորդ քայլը ֆայլ ավելացնելն է, որում և կգրենք ծրագիրը: Մկնիկի աջ կոճակով սեղմենք **Source Files** պանակի վրա և ընտրենք **Add > New Item...** :

5. Հայտնված պատուհանի միջոցով կարելի է նախագծին ֆայլեր ավելացնել: Այժմ ձախ ցանկից ընտրենք **Code** բաժինը և այնուհետև աջից ընտրեք **C++ File (.cpp)** – ն՝ ինչպես ցույց է տրված ստորև բերված նկարում: Ֆայլին տանք որևէ անուն և սեղմենք **Add** կոճակը:

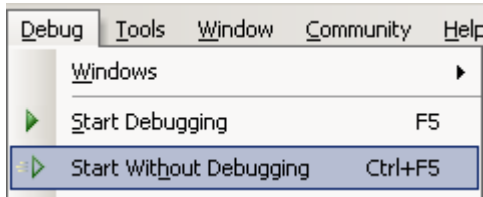



6. Ֆայլն ավելացնելուց հետո այն կհայտնվի **Source Files** պահանակի մեջ: Ֆայլի վրա մկնիկի ձախ կոճակի կրկնակի սեղմումով՝ ֆայլի պարունակությունը կբացվի նոր պատուհանի մեջ, և կկարողանանք աշխատել դրա հետ:

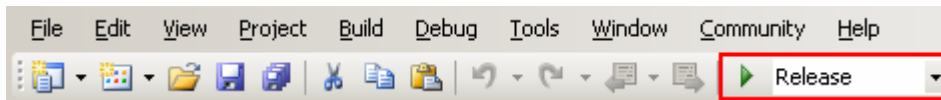


Ծրագրի կոմպիլացիա (Compilation) և աշխատեցում (Execution)

Եթե ծրագիրն արդեն պատրաստ է, և ցանկանում ենք աշխատեցնել այն, ապա մենյուից պիտի ընտրեք **Debug** ցանկը և բացված ցուցակից ընտրենք **Start Without Debugging** տողը, կամ պիտի պարզապես սեղմեք **Ctrl + F5** կոճակները:



Այս գործողությունը կարելի է կատարել նաև  սիմվոլը սեղմելով, սակայն նախորոք կողքի ցանկից ընտրելով **Release** տողը:



Եթե ծրագրում սխալներ չկան, ապա այն աշխատեցնելուց հետո կտեսնենք սպասվելիք արդյունքը, հակառակ դեպքում էկրանի ներքևի մասում կերևա հաղորդագրություն սխալի մասին (ֆայլի անունը, տողի համարը, սխալի բնույթը):

Բաժին 1.1

C++ - ի կառուցվածքը

Ծրագրավորման լեզու սովորելու ամենալավ եղանակն օրինակների վրա սովորելն է: Գրենք մեր առաջին ծրագիրը.

```
// Im arajin cragir@  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Barev!";  
    return 0;  
}
```

Barev!

Վերևի մասում գրված է մեր առաջին ծրագիրը, որը մենք կարող ենք գրել, օրինակ՝ **barev.cpp** անունով ֆայլի մեջ: Ներքևի մասում բերված է ծրագրի արդյունքն՝ այն աշխատեցնելուց հետո: Սա այն պարզագույն օրինակներից է, որոնք կարող են գրվել C++ լեզվով: Ծրագրի նպատակը «**Barev!**» բառը էկրանին տպելն է: Եկեք մանրամասն քննարկենք այս օրինակը:

```
// Im arajin cragir@
```

Սա մեկնաբանություն (comment) է: Բոլոր տողերը, որոնք սկսվում են // նշաններով, համարվում են մեկնաբանություններ և ծրագրի աշխատանքի վրա ոչ մի ազդեցություն չեն թողնում: Այս տիպի մեկնաբանությունը կարող է օգտագործվել ծրագրավորողի կողմից՝ կարճ բացատրությունների նպատակով: Մեր դեպքում այն օգտագործվում է որպես ծրագրի խորագիր:

```
#include <iostream>
```

Այն նախադասությունները, որոնք սկսվում են # նշանով, նախապրոցեսորի (preprocessor) հրամաններ են, որոնք օգտագործվում են թարգմանչին հրամաններ տալու համար: Մեր դեպքում **#include <iostream>** տողն ասում է թարգմանչի նախապրոցեսորին՝ ներառել **iostream** անունով ստանդարտ գրադարանը կամ, ինչ նույնն է, վերնագիր (**header**) ֆայլը: Այս ֆայլի մեջ նկարագրված է C++ - ի ստանդարտ մուտք-ելքի գրադարանը, որն օգտագործվում է ծրագրի հաջորդ տողերում:

```
using namespace std;
```

Այս տողը օգտագործվելու է մեր բոլոր ծրագրերում, սակայն այն բացատրվելու է գրքի վերջում: Այժմ, պարզապես նկատի ունենանք, որ այն կապված է նախորդ տողի հետ, որտեղ մենք ներառում էինք ստանդարտ մուտք-ելքի գրադարանը:

```
int main ()
```

Այս տողում մենք հայտարարում ենք `main` անունով ֆունկցիա: Սա այն ֆունկցիան է, որտեղից սկսում են աշխատել `C++` - ով գրված բոլոր ծրագրերը: Նշանակություն չունի, թե որտեղ այն կգրեք՝ ֆայլի սկզբում, մեջտեղում կամ վերջում, նրա պարունակությունը միշտ կաշխատի ծրագրի ամենասկզբում: Այդ պատճառով ակնհայտ է այն փաստը, որ `C++` - ով գրված բոլոր ծրագրերը միշտ պարունակում են `main` անունով ֆունկցիա:

`main` բառին անմիջապես հաջորդում են փակագծեր՝ `()`, քանի որ սա ֆունկցիա է: `C++` - ում բոլոր ֆունկցիաների անուններին անմիջապես հաջորդում են փակագծեր, որոնք անհրաժեշտության դեպքում կարող են պարունակել արգումենտներ: `main` - ում գրված ծրագիրը անմիջապես հաջորդում է ֆունկցիայի մարմինը, որը ներառված է ձևավոր `{ }` փակագծերի մեջ:

```
cout << "Barev!";
```

Այս հրամանը կատարում է մեր ծրագրի ամենակարևոր մասը: `cout` – ը `C++` - ի ստանդարտ ելքի հոսք է (`standard output stream`)՝ սովորաբար էկրանը: Այս տողով ելքի հոսքին ուղղարկում ենք սիմվոլների հաջորդականություն (մեր դեպքում՝ «`Barev!`»): `cout` – ը հայտարարված է `iostream` վերնագիր-ֆայլում, և որպեսզի կարողանանք օգտագործել այն, հարկավոր է «հրամայել» թարգմանչին՝ ներառել այդ ֆայլը:

Ուշադրություն դարձրեք, որ տողն ավարտվում է `;` սիմվոլով: Այս սիմվոլը ազդարարում է հրամանի ավարտի մասին, և այն պետք է կիրառվի յուրաքանչյուր հրամանի ավարտից հետո՝ `C++` - ով գրված բոլոր ծրագրերում (`C++` - ով ծրագրավորողների ամենահաճախ պատահող սխալներից մեկը հրամանից հետո `;` սիմվոլի բացակայությունն է):

```
return 0;
```

`return` հրամանը հանգեցնում է `main()` ֆունկցիայի ավարտին և վերադարձնում է այն կոդը, որը հաջորդում է հրամանին, մեր դեպքում՝ `0`: Ջրո վերադարձնելը նշանակում է, որ ծրագրի աշխատանքի ընթացքում ոչ մի սխալ չի եղել: Հետագա օրինակներում դուք կտեսնեք, որ `C++` - ով գրված բոլոր ծրագրերն ավարտվում են այս տողին նման տողով:

Ինչպես նկատեցիք, այս ծրագրի ոչ բոլոր տողերն են կատարում ինչ-որ գործողություն: Առաջին երկու տողերը պարունակում են միայն մեկնաբանություն (սրանք սկսվում են `//` սիմվոլներով) և նախապրոցեսորային հրաման (սրանք սկսվում են `#` սիմվոլով), հաջորդ տողում հայտարարվում է `main()` ֆունկցիան, և, վերջապես, վերջին տողերում գրված է `cout` – ին դիմելու հրամանը՝ սահմանափակված ձևավոր փակագծերով:

Ծրագիրը գրված է մի քանի տողով այն կարդալը և հասկանալը հեշտացնելու նպատակով: Նույն ծրագիրը կարող ենք գրել հետևյալ կերպ՝

```
#include <iostream>
using namespace std;
int main () { cout << "Barev!"; return 0; }
```

C++ - ում հրամանների տարանջատումը իրարից կատարվում է ; սիմվոլի միջոցով: Ստորև բերված է մեկ այլ ծրագիր՝ մի քանի հրամաններից կազմված:

```
// Im erkrord cragir@

#include <iostream>
using namespace std;

int main ()
{
    cout << "Barev! ";
    cout << "Yes grvac em C++ lezvov";
    return 0;
}

Barev! Yes grvac em C++ lezvov
```

Այս դեպքում օգտագործեցինք `cout <<` ֆունկցիան երկու անգամ՝ իրարից տարբեր երկու հրամանների մեջ: Մեկ անգամ ևս նշենք, որ ծրագրի բաժանումը մի քանի տողերի կատարված է միայն կարդալու հարմարավետության նպատակով: Միևնույն `main()` ֆունկցիան կարող էինք գրել հետևյալ կերպ՝

```
int main(){cout<<"Barev! ";cout<<"Yes grvac em C++ lezvov";return 0;}
```

Նույն ծրագիրը կարող ենք ներկայացնել նաև էլ ավելի շատ տողերի միջոցով՝

```
int main ()
{
    cout <<
        "Barev! ";
    cout
        << "Yes grvac em C++ lezvov";
    return 0;
}
```

Ծրագրի արդյունքը այս բոլոր դեպքերում կլինի նույնը:

Սակայն հիշենք, որ նախապրոցեսորի հրամանները (որոնք սկսվում են # սիմվոլով) չեն ենթարկվում այս կանոնին, որովհետև դրանք ծրագրի հրամաններ չեն: Սրանք տողեր են, որոնք կարդացվում և փոխարինվում են նախապրոցեսորի կողմից: Նախապրոցեսորի հրամանները պետք է գրված լինեն առանձին տողի վրա և չեն պահանջում ; սիմվոլը տողի ավարտին:

Մեկնաբանություններ (Comments)

Մեկնաբանությունները ծրագրի մասեր են, որոնք անտեսվում են թարգմանչի կողմից: Նրանք ոչինչ չեն անում: Մեկնաբանությունների իմաստն այն է, որ ծրագրավորողը, ցանկության դեպքում, ծրագրի մեջ կարողանա գրել հուշումներ և բացատրություններ:

C++ - ում կա մեկնաբանություններ գրելու երկու ձև՝

```
// տողային մեկնաբանություն
/* բլոկային մեկնաբանություն */
```

Տողային մեկնաբանությունն անտեսում է ամեն ինչ տրված տողում՝ սկսած // սիմվոլներից: Բլոկային մեկնաբանությունն անտեսում է /* */ սիմվոլների միջև ամեն ինչ:

Ավելացնենք որոշ մեկնաբանություններ մեր երկրորդ ծրագրին.

```
/* Im erkrord cragir@
   vorosh meknabanutiunnerov */

#include <iostream>
using namespace std;

int main ()
{
    cout << "Barev! ";      // tpum e Barev!
    cout << "Yes grvac em C++ lezvov "; // tpum e Yes grvac em C++
    lezvov
    return 0;
}

Barev! Es grvac em C++ lezvov
```

Եթե մեկնաբանությունները գրենք ծրագրի մեջ, առանց օգտագործելու վերը նշված ձևերից որևէ մեկը, ապա թարգմանիչը կընդունի գրվածը որպես տարբեր հրամաններ, և հավանաբար կտա հաղորդագրություն սխալի մասին:

Բաժին 1.2

Փոփոխականներ: Տվյալների տիպեր: Հաստատուններ

Նախորդ բաժնում քննարկված «Barev» ծրագրի օգտակար լինել-չլինելը հարցի տակ է: Մենք ստիպված եղանք գրել մի քանի տող կոդ, այնուհետև թարգմանել և միացնել ծրագրերը, որպեսզի էկրանի վրա մի նախադասություն ստանանք: Շատ ավելի հեշտ և արագ կլիներ ինքնուրույն գրել տրված նախադասությունը: Սակայն ծրագրավորումը չի սահմանափակվում մոնիտորի վրա տեքստ գրելով: Որպեսզի մի քիչ առաջ շարժվենք և սովորենք ծրագրեր գրել, որոնք օգտակար առաջադրանքներ կկատարեն և կխնայեն մեր ժամանակը, պետք է ծանոթանանք **փոփոխականի (variable)** գաղափարին:

Դիցուք՝ ես խնդրում եմ ձեզ մտքում պահել 5 թիվը, այնուհետև խնդրում եմ մտապահել նաև 2 թիվը: Դուք հենց նոր մտքում պահեցիք երկու արժեքներ: Այժմ, եթե ես խնդրեմ իմ անվանած առաջին թիվը ավելացնել 1 - ով, դուք պետք է մտքում ունենաք 6՝ (այսինքն՝ 5 + 1) և 2 թվերը: Հիմա կարող ենք այս թվերը իրարից հանել և ստանալ 4:

Այն գործողությունները, որոնք դուք կատարեցիք, շատ նման է նրան, ինչ կարող է կատարել համակարգիչը երկու փոփոխականների հետ: Այս նույն պրոցեսը կարելի է արտահայտել C++ լեզվով՝ հետևյալ հրամանների հերթականությամբ.

```
a = 5;  
b = 2;  
a = a + 1;  
ardyunq = a - b;
```

Սա, իհարկե, պարզագույն օրինակ է, քանի որ կային միայն երկու փոքր **ամբողջ (integer)** արժեքներ (ամբողջ թվեր), բայց ձեր համակարգիչը կարող է միաժամանակ պահել միլիոնավոր այդպիսի թվեր և դրանց հետ կատարել բարդ մաթեմատիկական գործողություններ:

Այսպիսով՝ մենք կարող ենք օգտագործել փոփոխականներ՝ նրանցում արժեքներ պահելու համար:

Ամեն փոփոխական պետք է ունենա **նույնարկիչ (identifier)**, որպեսզի այն հնարավոր լինի տարբերել մյուսներից (նախորդ օրինակում դրանք **a**, **b** և **ardyunq** - ն էին): Մենք կարող ենք փոփոխականներին տալ ցանկացած անուն, բայց այդ անունը պետք է լինի վավեր նույնարկիչ (բավարարի որոշակի պայմանների, որոնց մասին կխոսենք ստորև):

Նույնարկիչներ (Identifiers)

Վավեր նույնարկիչը մեկ կամ ավելի տառերի, թվերի և **_** սիմվոլների շարք է: Նույնարկիչի երկարությունը սահմանափակ չէ, սակայն որոշ թարգմանիչներ «ուշադրություն են դարձնում» նույնարկիչի միայն առաջին **32** սիմվոլների վրա (մնացածը անտեսվում է):

Նույնարկիչը չի կարող պարունակել դատարկ սիմվոլներ՝ **space** - ներ: Բացի դրանից, փոփոխականի նույնարկիչը չի կարող սկսվել թվով:

Նույնարկիչը նաև չի կարող համընկնել **C++** լեզվում սահմանված **բանալի-բառերի (Key Word)** հետ: Ներքոհիշյալ բառերը **C++** լեզվում սահմանված բանալի-բառեր են, և հետևաբար դրանք չի կարելի օգտագործել որպես նույնարկիչներ.

`asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t`

Բացի սրանցից, որոշ օպերատորների հետևյալ այլընտրանքային ներկայացումները նույնպես չի կարելի օգտագործել որպես նույնարկիչներ.

`and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq`

Շատ կարևոր է հիշել, որ **C++** լեզվում մեծատառերի և փոքրատառերի միջև տարբերություն է դրվում. այսինքն՝ օրինակ **ARDYUNQ** և **ardyunq** նույնարկիչները միմյանցից տարբեր են:

Տվյալների տիպեր (Data Types)

Ծրագրեր գրելիս մենք փոփոխականները պահում ենք համակարգչի հիշողության մեջ, բայց համակարգիչը պետք է իմանա, թե ինչ արժեքներ ենք ուզում պահել, քանզի պարզ փոքր թիվը, մեկ տառը և շատ մեծ թիվը միևնույն չափի հիշողություն չեն զբաղեցնում:

Մեր համակարգիչների հիշողությունը համակարգված է բայթերով: Բայթը հիշողության փոքրագույն քանակն է, որը կարող ենք օգտագործել: Մեկ բայթը կարող է պարունակել ինֆորմացիայի համեմատաբար փոքր քանակություն՝ սովորաբար **0 - 255** միջակայքի որևէ ամբողջ թիվ կամ միակ սիմվոլ (տառ): Բայց, բացի սրանցից, համակարգիչը կարող է աշխատել տվյալների ավելի բարդ տիպերի հետ, որոնք առաջանում են մի քանի բայթեր միավորելով. օրինակ՝ երկար թվերի հետ: Ներքևում բերված է **C++** լեզվում տվյալների հիմնական տիպերի ցուցակը, ինչպես նաև այն միջակայքերը, որոնցում նրանք կարող են արժեքներ ընդունել:

Տվյալների տիպեր

Անուն	Բայթեր	Բացատրություն	Միջակայք*
char	1	մեկ սիմվոլ կամ 8 բիթ երկարությամբ ամբողջ թիվ	-128 – ից 127 0 – ից 255
short	2	16 բիթ երկարությամբ ամբողջ թիվ	-32768 - ից 32767 0 - ից 65535
long	4	32 բիթ երկարությամբ ամբողջ թիվ	-2147483648 – ից 2147483647 0 - ից 4294967295
int	*	Ամբողջ թիվ: Երկարությունը կախված է այն օպերացիոն համակարգից և համակարգչից, որի վրա աշխատում ենք: MSDOS համակարգերի վրա դրա երկարությունը 16 բիթ է, իսկ Windows 9x/2000/NT , *nix և նմանատիպ համակարգերի դեպքում՝ 32 բիթ:	Տես short և long տիպերի միջակայքերը
float	4	Լողացող կետով թվեր: Կոտորակային թվերն են՝ 124.46878654	3.4e +/- 38 (7 նիշ)
double	8	Կրկնակի ճշտության լողացող կետով թվեր	1.7e +/- 308 (15 նիշ)
long double	10	Կրկնակի ճշտության լողացող կետով երկար թվեր	1.2e +/- 4932 (19 նիշ)
bool	1	Բուլյան արժեքներ: Կարող է ընդունել միայն երկու հնարավոր արժեք՝ true (ճիշտ) կամ false (ոչ ճիշտ)	true կամ false
wchar_t	2	Լայն սիմվոլ: Ստեղծվել է վերջերս, որպեսզի կարողանանք պահել երկու բայթանի սիմվոլներ (ոչ միայն անգլերեն տառեր, այլև հայերեն, արաբերեն...): Սա նոր ստանդարտ է, այդ պատճառով դեռևս ոչ բոլոր թարգմանիչներն ունեն այս տիպը	լայն սիմվոլ

*) Եթե տրված է երկու միջակայք, ապա առաջինը **նշանով** տարբերակն է, մյուսը՝ **առանց նշանի**:

Բացի այս տիպերից, կան նաև **ցուցիչներ (pointers)** և **դաստարկ (void)** տիպեր, որոնց կծանոթանանք հաջորդ գլուխներում:

Փոփոխականների հայտարարումը (Declaration of Variables)

Որպեսզի կարողանանք C++ լեզվում օգտագործել որևէ փոփոխական, պետք է այն նախապես հայտարարենք՝ նշելով նրա տեսակը: Գրելաձևը հետևյալն է. սկզբում գրում ենք տեսակը, ինչպիսին ուզում ենք լինի փոփոխականը (օրինակ՝ **int**, **short**, **float** ...), այնուհետև՝ փոփոխականի վավեր նույնարկիչ: Օրինակ՝

```
int a;
float mynumber;
```

Սրանք փոփոխականների ճիշտ հայտարարություններ են: Առաջինը հայտարարում է **int** տիպի փոփոխական՝ **a** նույնարկիչով (**անունով**): Երկրորդը հայտարարում է **float** տիպի փոփոխական՝ **mynumber** նույնարկիչով: Հայտարարվելուց հետո **a** և **mynumber** փոփոխականները կարող են օգտագործվել:

Եթե անհրաժեշտ է հայտարարել նույն տիպի մի քանի փոփոխական, կարող եք դրանք գրել մեկ տողի վրա՝ նրանց նույնարկիչները միմյանցից բաժանելով ստորակետերով: Օրինակ՝

```
int a, b, c;
```

տողը հայտարարում է **int** տիպի երեք փոփոխականներ (**a**, **b** և **c**) և ունի ճիշտ նույն իմաստը, ինչ հետևյալը՝

```
int a;
int b;
int c;
```

Ամբողջ տեսակները (**char**, **short**, **long** և **int**) կարող են լինել **նշանով (signed)** կամ **առանց նշանի (unsigned)**՝ կախված նրանից, թե թվերի ինչ միջակայք ենք ուզում ներկայացնել: Փոփոխականի հայտարարության մեջ տիպից առաջ գրում ենք **signed** կամ **unsigned**: Օրինակ՝

```
unsigned short MardkancQanak;           //mardkanc qanak@ misht drakan e
signed int AmsakanHashvekshir;         //hashvekshir@ karox e line1 < 0
```

Եթե չենք նշում **signed** կամ **unsigned**, ապա լռությամբ ընդունվում է **signed**. այսինքն՝ երկրորդ հայտարարության փոխարեն կարող էինք գրել՝

```
int AmsakanHashvekshir;
```

որը կունենար ճիշտ նույն նշանակությունը:

Որպեսզի ցուցադրենք, թե գործնականում փոփոխականների հայտարարումը ինչ է իրենից ներկայացնում, դիտարկենք այս բաժնի սկզբում բերված մտավոր հաշվարկի ներկայացումը C++ - ի կոդի տեսքով.

```
// ashxatanq popoxakanneri het

#include <iostream>
using namespace std;

int main ()
{
    // haytararum enq popoxakanner:
    int a, b;
    int ardyunq;

    // ashxatum enq dranc het:
    a = 5;
    b = 2;
    a = a + 1;
    ardyunq = a - b;

    // ekrani vra tpum enq ardyunq@:
    cout << ardyunq;

    // kangnecnum enq &ragir@:
    return 0;
}
```

4

Անհանգստանալու հարկ չկա, եթե որոշ քայլեր մի քիչ տարօրինակ են թվում: Այդ ամենին ավելի մանրամասն կձանոթանաք հաջորդ գլուխներում:

Փոփոխականների սկզբնարժեքավորում (Initialization of Variables)

Երբ հայտարարում ենք որևէ փոփոխական, դրա արժեքը սկզբում որոշված չէ: Եթե ուզում ենք, դեռևս հայտարարման ժամանակ, փոփոխականին արժեք տալ, գրում ենք հետևյալ կերպ.

տիպ նույնարկիչ = արժեք ;

Օրինակ՝ եթե ուզում ենք հայտարարել `int` տիպի `a` անունով փոփոխական և նրան տալ `0` արժեքը, կգրենք այսպես՝

```
int a = 0;
```

Բացի սրանից, կա նաև մեկ այլ տարբերակ՝

տիպ նույնարկիչ (արժեք) ;

Օրինակ՝

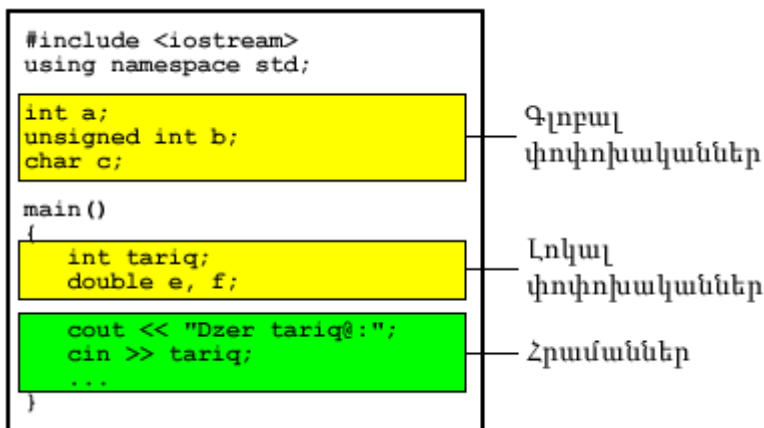
```
int a ( 0 );
```

C++ լեզվում այս երկուսը ճիշտ նույն նշանակությունն ունեն:

Փոփոխականների տեսանելիության տիրույթ (Scope of Variables)

Ինչպես ասացինք՝ բոլոր փոփոխականները, որոնք ուզում ենք օգտագործել, պետք է նախապես հայտարարված լինեն: C++ լեզվում փոփոխականները կարելի է հայտարարել կոդի մեջ՝ ցանկացած տեղում:

Սակայն խորհուրդ է տրվում փոփոխականների հայտարարությունն անջատել կոդի մնացած մասից:



Գլոբալ փոփոխականներին (global variables) կարելի է դիմել կոդի մեջ՝ ցանկացած տեղում՝ իր հայտարարությունից հետո:

Լոկալ փոփոխականների (local variables) տեսանելիության տիրույթը սահմանափակված է կոդի այն հատվածով, որում նրանք հայտարարված են եղել: Եթե նրանք հայտարարված են որևէ ֆունկցիայի սկզբում, ապա նրանց տեսանելիության տիրույթը այդ ֆունկցիան է: Սա նշանակում է, որ մյուս ֆունկցիաներում նրանք տեսանելի չեն (նրանց չի կարելի դիմել):

Հաստատուններ (Constants)

Հաստատունը արտահայտություն է, որը ունի ֆիքսած արժեք:

Ամբողջ թվեր (Integer Numbers)

```
1776
707
-273
```

սրանք մեզ լավ հայտնի 10 - ական համակարգի թվային հաստատուններ են:

Բացի 10 - ական համակարգի թվերից, C++ - ը թույլ է տալիս օգտագործել նաև 8 - ական համակարգի (octal) և 16 - ական համակարգի (hexadecimal) թվեր:

Որպեսզի թարգմանչին հասկացնենք, որ թիվը 8 - ական է, նրանից առաջ գրում ենք 0, իսկ 16 - ականներից առաջ՝ 0x: Օրինակ հետևյալ երեք արտահայտությունները միմյանց համարժեք են.

```
75          // 10-akan
0113       // 8-akan
0x4b       // 16-akan
```

Լողացող կետով թվեր (Floating Point Numbers)

Կարող են պարունակել 10 - ական թիվ, e սիմվոլ (սա նշանակում է՝ անգամ 10 - ի X աստիճան, որտեղ X - ը հաջորդող ամբողջ թիվ է):

```
3.14159    // 3.14159
6.02e23    // 6.02 x 1023
1.6e-19    // 1.6 x 10-19
3.0        // 3.0
```

Առաջին թիվը π - ն է, երկրորդը՝ Ավոգադրոյի հաստատունը, հաջորդը՝ էլեկտրոնի լիցքը, վերջինը՝ 3 բնական թիվը՝ արտահայտած որպես լողացող կետով թիվ:

Միավորներ և տողեր (Characters and strings)

Կան նաև ոչ թվային հաստատուններ՝

```
'z'
'p'
"Barev"
"Inchpes es?"
```

Առաջին երկու արտահայտություններն առանձին սիմվոլներ են, իսկ մյուս երկուսն իրենցից ներկայացնում են տողեր: Նկատենք, որ միայնակ սիմվոլը վերցնում ենք՝ սիմվոլների միջև, իսկ տողերը՝ " սիմվոլների միջև:

Միավորային և տողային հաստատունները ունեն որոշակի առանձնահատկություններ: Ներքևում բերված է հատուկ սիմվոլների ցուցակը.

\n	նոր տող
\r	Enter կոճակի կողը
\t	տաբուլացիա
\v	ուղղահայաց տաբուլացիա
\b	ջնջել մեկ սիմվոլ
\f	նոր էջ

<code>\a</code>	զանք
<code>\'</code>	' սիմվոլը
<code>\"</code>	" սիմվոլը
<code>\?</code>	? սիմվոլը
<code>\\</code>	\ սիմվոլը

Օրինակ՝

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Այս կոդերը ավելի պարզ կդառնան հաջորդ բաժիններում քննարկված օրինակների միջոցով:

Տողային հաստատունները կարող են ներկայացված լինել մեկից ավելի տողերի վրա: Դրա համար սողի վերջում դնում ենք `\` նշանը և շարունակում հաջորդ տողից.

```
"naxadasutyun` artahaytva& \
erku toxi vra"
```

Նշանակված հաստատուններ (Defined Constants) (`#define`)

Դուք կարող եք ինքնուրույն նշանակել հաստատուններ այն արտահայտությունների համար, որոնք հաճախ եք օգտագործում՝ `#define` հրամանի միջոցով: Գրելաձևը հետևյալն է.

```
#define նույնարկիչ արժեք
```

Օրինակ՝

```
#define PI 3.14159265
#define NORTOX '\n'
#define LAYNUTYUN 100
```

Սրանք նշանակում են 3 նոր հաստատուններ: Նշանակելուց հետո դրանք կարելի է օգտագործել կոդի մեջ՝ ինչպես ցանկացած այլ հաստատուն: Օրինակ՝

```
shrjanagic = 2 * PI * r;
cout << NORTOX;
```

Իրականում, միակ բանը, ինչ անում է համակարգիչը `#define` հրամանը տեսնելիս այն է, որ դրանից հետո հանդիպող *նույնարկիչ* բառը փոխարինում է *արժեք* արտահայտությամբ:

Հայտարարված հաստատուններ (Declared constants) (const)

const նշանաբանի միջոցով կարելի է հայտարարել հաստատուններ: Դա արվում է ճիշտ նույն ձև, ինչպես փոփոխականների դեպքում: Օրինակ՝

```
const int laynutyun = 100;  
const char tabulacia = '\\t';
```

Ստացված **laynutyun** և **tabulacia** հաստատունները կարելի է օգտագործել արտահայտությունների մեջ այնպես, ինչպես օգտագործում ենք փոփոխականները: Միակ բացառությունն այն է, որ հաստատունի արժեքը փոփոխող արտահայտությունները թույլատրված չեն:

Բաժին 1.3 Օպերատորներ

Այժմ, երբ արդեն ծանոթ ենք փոփոխականների և հաստատունների գաղափարին, կարող ենք աշխատել նրանց հետ: Այդ նպատակով C++ - ը մեզ տրամադրում է օպերատորներ: Օպերատորը իրենից ներկայացնում է հատուկ սիմվոլներից կազմված բառ: Այդ սիմվոլները այբուբենի տառեր չեն, բայց ներկա են բոլոր ստեղծագործության վրա: Շատ կարևոր է լավ հիշել հիմնական օպերատորները, քանի որ սրանք կազմում են C++ - ի անբաժանելի մասը:

Վերագրում՝ =

Վերագրման օպերատորը թույլ է տալիս փոփոխականին արժեք վերագրել:

```
a = 5;
```

վերագրում է 5 ամբողջ թիվը **a** - ին: Հավասարության = նշանից ձախ գտնվող մասն ընդունված է անվանել **lvalue** (left value, թարգմանած՝ ձախ արժեք), իսկ աջ մասը՝ **rvalue** (right value, թարգմանած՝ աջ արժեք): Արտահայտության ձախ արժեքը՝ **lvalue** - ն, պետք է միշտ լինի փոփոխական, իսկ աջ արժեքը՝ **rvalue** - ն, կարող է լինել հաստատուն, փոփոխական և դրանց ինչ-որ գործողության կամ զուգակցման արդյունք:

Հարկ է նշել, որ վերագրման գործողությունը միշտ տեղի է ունենում աջից ձախ և ոչ հակառակը.

```
a = b;
```

a փոփոխականին վերագրում է **b** - ի արժեքը՝ հաշվի չառնելով տվյալ պահին **a** - ում պահվող արժեքը: Հիշեք նաև այն հանգամանքը, որ **a** - ին վերագրում ենք միայն **b** - ի արժեքը: **a** - ի և **b** - ի միջև ոչ մի կապ չի ստեղծվում, հետևաբար **b** - ի արժեքի հետագա փոփոխությունները ոչ մի ազդեցություն չեն ունենա **a** - ի վրա:

Քննարկենք հետևյալ օրինակը.

```
int a, b;      // a:? b:?
a = 10;       // a:10 b:?
b = 4;        // a:10 b:4
a = b;        // a:4 b:4
b = 7;        // a:4 b:7
```

Արդյունքում կստանանք, որ **a** - ն 4 է, իսկ **b** - ն՝ 7: **b** - ի վերջին՝ **b = 7** ձևափոխությունը ոչ մի ազդեցություն չունեցավ **a** - ի վրա՝ չնայած դրանից առաջ գրել էինք՝ **a = b**:

C++ - ի առավելությունն այլ ծրագրավորման լեզուների նկատմամբ այն է, որ վերագրման գործողությունը կարող է օգտագործվել որպես աջ արժեք՝ **rvalue** (կամ որպես **rvalue** - ի մաս): Օրինակ՝

$a = 2 + (b = 5);$

համարժեք է հետևյալին՝

$b = 5;$

$a = 2 + b;$

ինչը նշանակում է՝ սկզբից վերագրել b – ին 5 , իսկ հետո a - ին վերագրել 2 - ին գումարած արդեն վերագրված b – ն, այսինքն՝ 5 : Հետևաբար ճիշտ է նաև հետևյալ արտահայտությունը՝

$a = b = c = 5;$

որտեղ 5 թիվը վերագրվում է բոլոր երեք՝ a , b և c փոփոխականներին:

Թվաբանական գործողություններ՝ +, -, *, /, %

C++ - ում սպասարկվում են հետևյալ հինգ գործողությունները՝

+	գումարում
-	հանում
*	բազմապատկում
/	բաժանում
%	մոդուլ

Գումարման, հանման, բազմապատկման և բաժանման օպերատորների օգտագործումն առանձնահատուկ բացատրության կարիք չունի, քանի որ տարրական մաթեմատիկայից այն արդեն հայտնի է:

Միայն մոդուլ – ը % կարող է ձեռք համար անհասկանալի լինել: **մոդուլ** - ը այն գործողությունն է, որը վերադարձնում է երկու ամբողջ թվերի բաժանման հետևանքից ստացված մնացորդը: Օրինակ՝ եթե գրենք $a = 11 \% 3$, ապա a - ի արժեքը կլինի 2 , քանի որ, եթե բաժանենք 11 - ը 3 - ի, կստանանք 2 մնացորդ:

Բարդ վերագրման օպերատորներ՝ +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

Այս օպերատորների միջոցով հնարավորություն է տրվում փոխել փոփոխականի արժեքը մեկ գործողության միջոցով: Օրինակ՝

```
value += increase;
```

նույնն է, ինչ

```
value = value + increase;
```

```
a -= 5;
```

նույնն է, ինչ

```
a = a - 5;
```

```
a /= b;
```

```
նույնն է, ինչ
a = a / b;
```

```
price *= units + 1;
նույնն է, ինչ
price = price * (units + 1);
```

Նույնը բոլոր մնացած օպերատորների համար:

Մեծացման և փոքրացման օպերատորներ՝ ++, --

Մյուս օպերատորները, որոնց միջոցով կարող ենք փոխել փոփոխականի արժեքը մեկ գործողության միջոցով, մեծացման ++ և փոքրացման -- օպերատորներն են: Նրանք մեծացնում կամ փոքրացնում են փոփոխականի արժեքը 1 - ով: Նրանք համարժեք են += 1 և -= 1 գործողություններին: Հետևաբար՝

```
a++;
a += 1;
a = a + 1;
```

արտահայտություններն իրար համարժեք են՝ բոլորը մեծացնում են **a** - ի արժեքը 1 - ով:

Տրված օպերատորները կարող են օգտագործվել և՛ որպես **prefix**, և՛ որպես **suffix** (postfix): Դա նշանակում է, որ սրանք կարող են գրվել փոփոխականից հետո՝ **a++** (suffix), կամ առաջ՝ **++a** (prefix): Չնայած որ **a++** և **++a** պարզ արտահայտություններն ունեն միևնույն նշանակությունը, սակայն այլ դեպքերում, որտեղ *մեծացման* և *փոքրացման* գործողությունները հանդիսանում են մեկ այլ (ավելի բարդ) գործողության մի մաս, նրանք կարող են ունենալ բացարձակապես տարբեր նշանակություն: Այն դեպքում, երբ մեծացման օպերատորը օգտագործվում է որպես **prefix**՝ **++a**, **a** - ի արժեքը մեծացվում է մինչև (բարդ) արտահայտության հաշվումը: Այսինքն բարդ արտահայտության մեջ **a** - ն մասնակցում է արդեն մեծացված արժեքով: Իսկ եթե մեծացման օպերատորը օգտագործվում է, որպես **suffix**՝ **a++**, **a** - ի արժեքը մեծացվում է ընդհանուր արտահայտության արժեքի հաշվումից հետո: Տարբերությունը տեսնելու համար դիմենք հետևյալ օրինակին.

Օրինակ 1.

```
B = 3;          // A: ? B: 3
A = ++B;       // A: 4 B: 4
```

Օրինակ 2.

```
B = 3;          // A: ? B: 3
A = B++;       // A: 3 B: 4
```

Համեմատության օպերատորներ՝ ==, !=, >, <, >=, <=

Երկու արտահայտություններ իրար հետ համեմատելու համար մենք կարող ենք օգտագործել համեմատության օպերատորները: Համեմատության գործողության արդյունքը լինելու է **bool** տիպի, որը կրնա լինի **true** (ճշմարիտ) կամ **false** (ոչ ճշմարիտ) արժեքները՝ կախված արտահայտությունների համեմատությունից:

Ստորև բերված է համեմատության օպերատորների ցուցակը.

==	հավասար
!=	տարբեր
>	մեծ
<	փոքր
>=	մեծ կամ հավասար
<=	փոքր կամ հավասար

Ստորև բերված են որոշ օրինակներ.

(7 == 5)	վերադարձնելու է false
(5 > 4)	վերադարձնելու է true
(3 != 2)	վերադարձնելու է true
(6 >= 6)	վերադարձնելու է true
(5 < 5)	վերադարձնելու է false

Թվային հաստատունների փոխարեն կարող ենք օգտագործել նաև ցանկացած <<ճիշտ>> արտահայտություն, օրինակ՝ փոփոխականներ: Դիցուք՝ **a = 2**, **b = 3** և **c = 6**: Այդ դեպքում.

(a == 5)	վերադարձնելու է false
(a*b >= c)	վերադարձնելու է true քանի, որ $2*3 \geq 6$
(b+4 > a*c)	վերադարձնելու է false քանի, որ $(3+4) < 2*6$
((b=2) == a)	վերադարձնելու է true

Բայց զգույշ եղեք: = օպերատորը նույնը չէ, ինչ == օպերատորը. առաջինը վերագրման օպերատոր է, երկրորդը՝ համեմատության: Այդ է պատճառը, որ վերջին ((b=2) == a) օրինակում սկզբից **b** - ին վերագրվում է **2**՝ (b=2), իսկ հետո նոր **b**-ն համեմատվում է **a** - ի հետ: Քանի որ **a** - ն **2** է, համեմատման արդյունքը ստացվեց **true**:

Տրամաբանական օպերատորներ՝ !, &&, ||

! օպերատորը համարժեք է բուլյան ժխտման օպերատորին և նա պահանջում է միայն մեկ օպերանդ՝ գրված նրանից աջ: Փաստորեն այս օպերատորի իմաստը արժեքի շրջումն (ժխտում) է. վերադարձնել **false**, եթե օպերանդը **true** է, և **true**, եթե օպերանդը **false** է: Սա նույնն է, թե ասել, որ այս օպերատորը վերադարձնում է օպերանդի արժեքի հակադիրը: Օրինակ.

!(5 == 5)	վերադարձնում է false , որովհետև նրանից աջ գտնվող արտահայտությունը (5 == 5) true է
!(6 <= 4)	վերադարձնում է true , որովհետև 6 <= 4 false է
!true	վերադարձնում է false
!false	վերադարձնում է true

&& և **||** տրամաբանական օպերատորներն օգտագործվում են, երբ գնահատվում են երկու արտահայտություններ՝ մեկ պատասխան ստանալու նպատակով: Մրանք համապատասխանում են համապատասխանաբար բուլյան *կոնյունկցիա* և բուլյան *դիզյունկցիա* ֆունկցիաներին: Գործողության արդյունքը կախված է երկու օպերանդներից.

Առաջին օպերանդ a	Երկրորդ օպերանդ b	Արդյունք a && b	Արդյունք a b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Օրինակ.

((5 == 5) && (3 > 6)) վերադարձնում է **false** (true && false)

((5 == 5) || (3 > 6)) վերադարձնում է **true** (true || false)

Պայմանական օպերատոր՝ ?

Պայմանական օպերատորը գնահատում է տրված արտահայտությունը և կախված նրա արժեքից՝ **true** կամ **false**, վերադարձնում է համապատասխան արժեք: Այս օպերատորի գրելաձևը հետևյալն է՝
պայման ? արդյունք1 : արդյունք2

Եթե *պայման* - ը **true** է, արտահայտությունը կվերադարձնի *արդյունք1*, հակառակ դեպքում՝ *արդյունք2*:

(7==5) ? 4 : 3	կվերադարձնի 3 , քանի որ 7 - ը հավասար չէ 5 - ի:
(7==5+2) ? 4 : 3	կվերադարձնի 4 , քանի որ 7 - ը հավասար է 5+2 - ի:
(5>3) ? a : b	կվերադարձնի a , քանի որ 5 - ը մեծ է 3 - ից:
(a>b) ? a : b	կվերադարձնի մեծագույնը՝ a - ն կամ b - ն:

Բիթային օպերատորներ՝ &, |, ^, ~, <<, >>

Բիթային օպերատորները ձևափոխում են փոփոխականները՝ աշխատելով նրանց արժեքի երկուական ներկայացման հետ:

Օպերատոր	Անուն	Բացատրություն
&	AND	բիթային ԵՎ
	OR	բիթային ԿԱՍ
^	XOR	բիթային բացառիկ ԿԱՍ
~	NOT	Լրացում
<<	SHL	ձախ տեղաշարժ
>>	SHR	աջ տեղաշարժ

Տիպերի ձևափոխման օպերատորներ

Տիպերի ձևափոխման օպերատորները թույլ են տալիս մի տիպը վերածել մեկ այլ տիպի: **C++** - ում կան մի քանի եղանակներ այդ բանն իրագործելու համար: Այդ ձևերից ամենատարածվածը վերածվող արտահայտության նախորդումն է փակագծերի մեջ գրված նոր տիպով.

```
int i;
float f = 3.14;
i = (int) f;
```

Այս ծրագիրը **float** տիպի **3.14** թիվը վերածում է **ամբողջ` int** տիպի: Այստեղ որպես տիպի ձևափոխման օպերատոր հանդես է գալիս (**int**) - ը: Այս նույն գործողությունը կատարելու մյուս եղանակը **կառուցիչի (constructor)** օգտագործումն է: Այս դեպքում գրում ենք նոր տիպը և այնուհետև փակագծերի մեջ գրել ձևափոխվող արտահայտությունը.

```
i = int ( f );
```

Այս դեպքում ստեղծվում է նշված տիպի նոր օբյեկտ: Ներկայացված երկու ձևափոխման եղանակներից բացի կան նաև այլ եղանակներ, որոնք կներկայացվեն հաջորդ բաժիններում:

sizeof() օպերատորը

Այս օպերատորն ընդունում է մեկ արգումենտ, որը կարող է լինել կամ փոփոխականի տիպ, կամ էլ փոփոխական, և վերադարձնում է այդ օբյեկտի կամ տիպի չափը` արտահայտված բայթերով (bytes).

```
a = sizeof (char);
```

Այս տողը **a** – ին կվերագրի **1**, քանի որ **char** տիպը զբաղեցնում է 1 բայթ:

sizeof - ի կողմից վերադարձվող թիվը հաստատուն է, այսինքն այն արդեն որոշված է ծրագրի աշխատանքից առաջ (դեռևս թարգմանության ժամանակ):

Օպերատորների նախապատվությունը

Մի քանի օպերատորներից բարդ արտահայտություններ կազմելիս կարող է առաջանա հարց, թե ինչ հերթականությամբ կկատարվեն այդ գործողությունը: Օրինակ՝ $a = 5 + 7 \% 2$ արտահայտության մեջ կարող է կասկած առաջանալ, թե արդյոք այս արտահայտությունը կդիտարկվի որպես $a = 5 + (7 \% 2)$ արտահայտություն՝ 6 արդյունքով, թե՛ $a = (5 + 7) \% 2, 0$ արդյունքով: Ճիշտ պատասխանն առաջինն է՝ 6: C++ - ում գոյություն ունեն օպերատորների (ոչ միայն թվաբանական, այլ բոլոր օպերատորների) նախապատվությունների մակարդակներ, ըստ որոնց որոշվում է գործողությունների կատարման հերթականությունը: Ստորև բերված է այդ ցանկը.

Նախ.	Օպերատոր	Բացատրություն	Ասոցիատիվություն
1	::	տեսանելիության տիրույթ	ձախ
2	() [] -> . sizeof		ձախ
3	++ --	մեծացում / փոքրացում	աջ
	~	բիթային (bitwise) լրացում	
	!	ունար ՈՉ	
	& *	հասցեավորում և հետհասցեավորում	
	(type)	տիպի ձևափոխություն	
	+ -	ունար գումար, տարբերություն	
4	* / %	թվաբանական գործողություններ	ձախ
5	+ -	թվաբանական գործողություններ	ձախ
6	<< >>	բիթային տեղաշարժ	ձախ
7	< <= > >=	համեմատության օպերատորներ	ձախ
8	== !=	համեմատության օպերատորներ	ձախ
9	& ^	բիթային օպերատորներ	ձախ

10	&&	տրամաբանական օպերատորներ	ձախ
11	? :	պայմանական օպերատոր	աջ
12	= += -= *= /= %= >>= <<= &= ^= =	վերագրման օպերատորներ	աջ
13	,	ստորակետ՝ տարանջատիչ	աջ

Ասոցիատիվություն - ը ցույց է տալիս, թե օպերատորների միևնույն նախապատվությունների դեպքում առաջինը որ օպերատորը պետք է կատարվի՝ աջակողմյանը, թե ձախակողմյանը:

Բոլոր այս օպերատորները կարող են օգտագործվել կամ դառնալ օգտագործելու համար ավելի հարմար՝ փակագծերի միջոցով: Օրինակ՝

$a = 5 + 7 \% 2;$

արտահայտության փոխարեն խորհուրդ է տրվում գրել

$a = 5 + (7 \% 2);$

կամ

$a = (5 + 7) \% 2;$

կախված նրանից, թե որ գործողությունն ենք ուզում կատարվի ավելի շուտ:

Այդ պատճառով, եթե դուք ուզում եք գրել բարդ արտահայտություն և համոզված չեք, թե որ գործողությունը կկատարվի ավելի շուտ, միշտ օգտագործեք փակագծեր:

Բաժին 1.4

Հաղորդակցություն օգտագործողի հետ

Հիմնական սարքերը, որոնց միջոցով աշխատում են համակարգչի հետ, մոնիտորն ու ստեղնաշարն են: Ստեղնաշարը հիմնական մուտքի սարքն է, իսկ մոնիտորը՝ հիմնական ելքի:

C++ - ի **iostream** գրադարանում ստանդարտ մուտքի-ելքի գործողությունները կատարվում են **cin** և **cout** հոսքերի միջոցով: Գոյություն ունեն նաև երկու հավելյալ հոսքեր՝ **cerr** և **clog**, որոնք նախատեսված են ծրագրում առաջացած սխալներ մոնիտորի վրա ցույց տալու համար:

Օգտագործելով այս երկու հոսքերը՝ հնարավոր է աշխատել օգտագործողի հետ՝ նրան ցույց տալ ինֆորմացիա և նրանից ստանալ հրամաններ:

Ելք (cout)

cout հոսքը օգտագործվում է << օպերատորի հետ.

```
cout << "elqi naxadasutyun"; // tpum e elqi naxadasutyun ekrani vra
cout << 120;                  // tpum e 120 tiv@ ekrani vra
cout << x;                    // ekrani vra tpum e x popoxakani arjeq@
```

<< օպերատորը հայտնի է որպես **տեղադրման օպերատոր**, քանի որ այն ինֆորմացիան դնում է հոսքի մեջ: Այս օրինակում այն տեղադրում է **"elqi naxadasutyun"** տողային հաստատունը, 120 թվային հաստատունը և x փոփոխականի արժեքը **cout** ելքի հոսքի մեջ: Նկատենք, որ **"elqi naxadasutyun"** - ը տեղադրված է " սիմվոլների միջև, քանզի այն տողային հաստատուն է: Տողային հաստատունները միշտ պետք է գրվեն " սիմվոլների միջև, որպեսզի տարբերվեն փոփոխականներից:

```
cout << "barev";           // ekranin tpum e barev
cout << barev;             // ekranin tpum e barev popoxakani parunakutyun@
```

Տեղադրման օպերատորը << մեկ հրամանի մեջ կարող է օգտագործվել 1 - ից ավելի անգամ.

```
cout << "barev, " << "Yes " << "C++i naxadasutyun em";
```

Վերջին տողը կտալի **"Barev, Yes C++i naxadasutyun em"** տողը: Մի հրամանի մեջ մի քանի << օպերատոր ունենալու առավելությունը լավ է երևում հետևյալ օրինակում՝

```
cout << "Barev, Yes &nvel em " << taretiv << " tvin yev hima " <<
tariq << " tarekan em";
```

Եթե ենթադրենք, որ **taretiv** փոփոխականը պարունակում է **1988** թիվը, իսկ **tariq** փոփոխականը՝ 16 թիվը, ապա այս տողը կտալի՝

Barev, Yes &nvel em 1988 tvin yev hima 16 tarekan em

Ուշադրություն դարձրեք, որ **cout** - ը ամեն ելքից հետո նոր տողի չի անցնում:

```
cout << "Sa naxadasutyun e.";  
cout << "Sa mek ayl naxadasutyun e.";
```

ցույց կտա՝

```
Sa naxadasutyun e.Sa mek ayl naxadasutyun e.
```

Այսպիսով, որպեսզի անցնենք նոր տողի, անհրաժեշտ է դնել նոր տողի անցնելու **\n** սիմվոլ:

```
cout << "Arajin naxadasutyun.\n " ;  
cout << "Yerkrord naxadasutyun.\nYerord naxadasutyun." ;
```

կառաջացնի հետևյալ ելքը.

```
Arajin naxadasutyun.  
Yerkrord naxadasutyun.  
Yerord naxadasutyun.
```

Բացի սրանից, նոր տողի է կարելի անցնել նաև **endl** հրամանի միջոցով: Օրինակ՝

```
cout << "Arajin naxadasutyun ." << endl ;  
cout << "Yerkrord naxadasutyun." << endl ;
```

կտայի՝

```
Arajin naxadasutyun .  
Yerkrord naxadasutyun.
```

Մուտք (cin)

C++ - ում ստանդարտ մուտքը ապահովվում է հոսքից հանելու **>>** օպերատորի և **cin** հոսքի օգնությամբ: Սրանց պետք է հետևի այն փոփոխականի անունը, որում ուզում ենք պահել ստացված ինֆորմացիան: Օրինակ՝

```
int tariq;  
cin >> tariq;
```

Սա հայտարարում է **tariq** փոփոխականը և սպասում, որ ստեղնաշարից տվյալներ մուտքագրվեն. ստացված տվյալները պահվում են **tariq** փոփոխականի մեջ:

cin - ը ստանում է ինֆորմացիա միայն **Enter** կոճակը սեղմելուց հետո:

Անհրաժեշտ է միշտ ուշադիր լինել այն փոփոխականի տիպի նկատմամբ, որի մեջ պետք է պահվի ստացված ինֆորմացիան: Եթե պահանջենք ամբողջ տիպի թիվ, ապա կատանանք ամբողջ տիպի թիվ, եթե պահանջենք սիմվոլային տող, կատանանք սիմվոլային տող:

```
// mutqi-yelqi orinak
#include <iostream>
using namespace std;

int main ()
{
    int i;
    cout << "Grir amboxch tiv: ";
    cin >> i;
    cout << "Du grecir " << i;
    cout << " yev nra krknapatik@ " << i*2 << " e.\n";
    return 0;
}
```

```
Grir amboxch tiv: 702
Du grecir 702 yev nra krknapatik@ 1404 e.
```

Ծրագրում սխալներ առաջանալու հիմնական պատճառը ծրագիրն օգտագործողն է, և քանի որ նրա հետ չի կարելի վարվել «Գլխից էլ պրծնենք, գլխացավից էլ» սկզբունքով, ապա ստիպված ենք հաշտվել այն գաղափարի հետ, որ հնարավոր է, որ օգտագործողը ծրագրին սխալ ծրագրեր տա (օրինակ՝ դու հարցնես տարիքը և սպասես, որ նա վերադարձնի ամբողջ տիպի թիվ, իսկ նա գրի իր անունը՝ սիմվոլների տող): Հիմա դեռ ստիպված կլինենք լիովին վստահել օգտագործողին, սակայն, շարունակելով ուսումնասիրել այս ձեռնարկը, շուտով կգտնենք այս խնդրի որոշակի լուծումներ:

cin - և **cout** - ի նման մեկ տողի վրա կարող է ունենալ 1 - ից ավելի >> օպերատոր՝

```
cin >> a >> b;
```

համարժեք է հետևյալին՝

```
cin >> a;
cin >> b;
```

Բաժին 2.1

Կառավարման համակարգեր (Control Structures)

Հաճախ անհրաժեշտ է լինում, որ ծրագրի որևէ մասը աշխատանքի ընթացքում կատարվի մի քանի անգամ կամ ընդհանրապես չկատարվի՝ կախված որոշակի պայմաններից: Այդ նպատակով **C++** - ում ստեղծել են կառավարման համակարգեր: Մինչև կառավարման համակարգերն ուսումնասիրելը քննարկենք մի նոր հասկացություն՝ **հրամանների բլոկ** (block of instructions): Հրամանների բլոկն իրենից ներկայացնում է հրամանների շարան՝ տարանջատած կետ-ստորակետ ; սիմվոլներով և սահմանափակված ձևավոր փակագծերով { }:

Կառավարման համակարգերի մեծ մասը, որոնք այժմ կուսումնասիրենք, որպես պարամետր կարող են ընդունել մեկ կամ ավելի հրամաններ: Այն դեպքում, երբ որպես պարամետր հանդես կգա մեկ հրաման, անհրաժեշտ չի լինի գրել այդ հրամանը ձևավոր փակագծերի մեջ: Իսկ եթե ցանկանանք գրել մի քանի հրաման,

ապա դրանք պարտադիր պետք է սահմանափակված լինեն ձևավոր փակագծերով՝ առաջացնելով հրամանների բլոկ:

Պայմանայան համակարգ if և else

Այս համակարգի տեսքը հետևյալն է.

```
if ( պայման ) մարմին
```

Եթե *պայման* -ի մեջ գրված արտահայտությունը կատարվում է (ճիշտ է), այսինքն՝ *պայման* - ը **true** է, ապա կկատարվի *մարմին* - ում գրված ծրագիրը, հակառակ դեպքում՝ *մարմին* - ում գրված ծրագիրը չի կատարվի:

Օրինակ՝ ստորև բերված ծրագիրը կտպի <<**x-@ 100 e**>> այն և միայն այն դեպքում, երբ **x** – ը իրոք լինի **100**.

```
if ( x == 100 )
    cout << "x-@ 100 e";
```

Եթե ցանկանանում ենք, որ *պայման* - ը **true** լինելու դեպքում կատարվի մեկից ավելի հրաման, ապա պետք է գրենք այդ հրամանները ձևավոր փակագծերի մեջ.

```
if ( x == 100 )
{
    cout << "x-@ ";
    cout << x << " e";
}
```

Կարող ենք նաև սահմանել գործողություն այն դեպքի համար, երբ *պայման* - ը **true** չէ՝ օգտագործելով **else** բանալի-բառը: Այն օգտագործվում է միայն **if** - ի հետ, և նրա տեսքն է.

```
if ( պայման ) մարմին1 else մարմին2
```

Օրինակ՝

```
if ( x == 100 )
    cout << "x-@ 100 e";
else
    cout << "x-@ 100 che";
```

Էկրանին տպում է <<**x-@ 100 e**>>, եթե **x** – ը իրոք **100** է, և <<**x-@ 100 che**>>՝ հակառակ դեպքում:

if + **else** կառուցվածքների միջոցով կարող ենք կազմել բարդ համակարգեր, օրինակ՝ որոշել ինչ-որ **x** փոփոխականի նշանը.

```
if ( x > 0 )
    cout << "x-@ drakan e";
else if ( x < 0 )
    cout << "x-@ bacasakan e";
else
    cout << "x-@ 0 e";
```

Բայց հիշեք, որ եթե դուք ուզում եք օգտագործել մեկից ավելի հրամաններ, միշտ սահմանափակեք այդ հրամանները ձևավոր փակագծերով.

```
if (x > 0)
    cout << "x-@ drakan e";
else if (x < 0) {
    cout << "x-@ ";
    cout << "bacasakan e";
}
else
    cout << "x-@ 0 e";
```

Կրկնվող համակարգեր կամ ցիկլեր (Repetitive Structures or Loops)

Ցիկլերի նպատակն է կրկնել *մարմին* - ը մի քանի անգամ կամ քանի դեռ պայմանը ճիշտ է:

while ցիկլը

Տեսքը.

```
while (պայման) մարմին
```

While թարգմանաբար նշանակում է՝ քանի դեռ: Այս ցիկլը կրկնում է *մարմին* - ը, քանի դեռ *պայման* - ը ճիշտ է:

Օրինակ՝ գրենք ծրագիր, որը կթվարկի տրված թվից սկսած և նրանից փոքր բոլոր դրական թվերը՝ օգտագործելով **while** ցիկլը.

```
#include <iostream>
using namespace std;
int main ()
{
    int n;
    cout << "Nermuceq skzbnakan tiv@ > ";
    cin >> n;
    while (n>0) {
        cout << n << ", ";
        --n;
    }
    cout << "VERJ!";
    return 0;
}
```

```
Nermucec skzbnakan tiv@ > 8
8, 7, 6, 5, 4, 3, 2, 1, VERJ!
```

Երբ ծրագիրը սկսում է աշխատել, այն օգտագործողին խնդրում է ներմուծել ինչ-որ սկզբնական թիվ: Այնուհետև սկսվում է **while** ցիկլը: Եթե օգտագործողի ներմուծած թիվը բավարարում է $n > 0$ (n - ը մեծ է 0 - ից) պայմանին, ապա հաջորդող հրամանների բլոկը կատարվում է այնքան անգամ, քանի դեռ $n > 0$ պայմանը ճիշտ է:

Վերը բերված ծրագրի ամբողջ պրոցեսը կարող է նկարագրվել հետևյալ կերպ՝ սկսած **main** ֆունկցիայից.

- 1. Օգտագործողը **n** - ին վերագրում է արժեք:
- 2. **while** հրամանը ստուգում է, թե արդյոք (**n > 0**). այստեղ հնարավոր է երկու դեպք.
 - **true**. կատարվում է **մարմին** - ը (քայլ **3** - րդ),
 - **false**. բաց է թողնվում մարմինը, և կատարվում է **5** - րդ քայլը:
- 3. Կատարվում է **մարմին** - ը.
`cout << n << ", ";`
`--n;`
(**n** - ը տպում է էկրանին, իսկ հետո նվազեցնում այն 1-ով):
- 4. Բլոկը ավարտվում է. վերադարձ **2**-րդ քայլին:
- 5. Կատարվում է բլոկից հետո եկող ծրագիրը՝ էկրանին տպվում է **<<VERJ! >>**, և ծրագիրն ավարտվում է:

Նշենք նաև այն հանգամանքը, որ ցիկլը պիտի ունենա ավարտ, այսինքն՝ ցիկլի **մարմին** - ում պետք է ներկա լինի **պայման** - ը **false** դարձնող ինչ-որ արտահայտություն, որն էլ և կստիպի ցիկլին ավարտվել: Մեր դեպքում, որպես այդպիսի արտահայտություն, հանդես է գալիս **<<--n;>>** հրամանը, որը ցիկլի որոշակի կրկնություններից հետո **n** - ը **0** է դարձնում, ինչը ստիպում է ցիկլին ավարտվել:

do - while ցիկլը

Տեսքը.

```
do մարմին while (պայման);
```

Այս ցիկլը լրիվ նույնն է, ինչ **while** ցիկլը, միայն այն տարբերությամբ, որ **պայման** - ը ստուգվում է ոչ թե ցիկլի սկզբում, այլ վերջում: Այս դեպքում, անկախ **պայման** - ի ճիշտ կամ սխալ լինելուց, **մարմին** - ը գոնե մեկ անգամ կատարվում է:

Օրինակ՝ հետևյալ ծրագիրը կլիստրի օգտագործողին մուտքագրել ինչ-որ մի թիվ այնքան անգամ, քանի դեռ մուտքագրված թիվը **0** չի: Ծրագիրը նաև կտալի բոլոր մուտքագրված թվերը.

```
#include <iostream.h>
int main ()
{
    unsigned long n;
    do {
        cout << "Mutqagrek tiv (0 avarti hamar): ";
        cin >> n;
        cout << "Duk mutqagrecik: " << n << "\n";
    } while (n != 0);
    return 0;
}
```

```
Mutqagrek tiv (0 avarti hamar): 12345
Duk mutqagrecik: 12345
```

```
Mutqagrek tiv (0 avarti hamar): 160277
Duk mutqagrecik: 160277
Mutqagrek tiv (0 avarti hamar): 0
Duk mutqagrecik: 0
```

do - while ցիկլը սովորաբար օգտագործվում է այն դեպքերում, երբ ցիկլի ավարտը որոշող արտահայտությունը որոշվում է ցիկլի ներսում՝ *մարմնում*: Մեր դեպքում ցիկլի ավարտը որոշող արտահայտությունը՝ **n** փոփոխականը, արժեք է ստանում ցիկլի *մարմնում* և այդ է պատճառը, որ մենք օգտագործեցինք **do - while** ցիկլը: Եթե այս ծրագրում օգտագործողը երբեք **0** չմուտքագրի, ապա ծրագիրը չի ավարտվի:

for ցիկլը

Տեսքը.

for (*սկզբնարժեքավորում; պայման; փոփոխություն*) *մարմին*;

Ինչպես և **while** – ում, սրա հիմնական ֆունկցիան *մարմինը* կրկնելն է, քանի դեռ *պայմանը* ճիշտ է: Բայց այս ցիկլը նաև թույլ է տալիս սահմանել **սկզբնարժեքավորում** և **փոփոխություն** օպերատորները:

Ահա թե ինչպես է այն աշխատում.

1. Կատարվում է *սկզբնարժեքավորում* օպերատորը: Սովորաբար այն օգտագործվում է փոփոխականի սկզբնարժեքավորման համար: Այս օպերատորը կատարվում է միայն մեկ անգամ:
2. Ստուգվում է *պայմանը*, և եթե այն **true** է, ապա ցիկլը շարունակվում է, հակառակ դեպքում՝ ավարտվում՝ բաց թողնելով *մարմինը*:
3. Կատարվում է *մարմինը*: Ինչպես միշտ, այն կարող է կամ մեկ հրամանից բաղկացած լինել, կամ էլ մի քանի հրամաններից՝ սահմանափակված ձևավոր փակագծերով:
4. Վերջապես, կատարվում է *փոփոխություն* օպերատորը, և ցիկլը վերադառնում է 2-րդ քայլին:

Օրինակ.

```
#include <iostream>
using namespace std;

int main ()
{
    int n;
    for (n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "VERJ!";
    return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, VERJ!
```

Սկզբնարժեքավորում և *փոփոխություն* օպերատորները պարտադիր չեն: Նրանք կարող են բաց թողնվել, բայց կետ-ստորակետերը պիտի մնան իրենց տեղում: Օրինակ՝ կարող ենք գրել `for(;n<10;)`, եթե մենք չենք ուզում իրականացնել որևէ սկզբնարժեքավորում և փոփոխություն, կամ՝ `for(;n<10;n++)`, եթե ուզում ենք սահմանել *փոփոխություն* օպերատորը, բայց չենք ուզում իրականացնել որևէ սկզբնարժեքավորում:

Օգտագործելով ստորակետ , օպերատորը՝ կարող ենք գրել մի քանի հրամաններ `for` – ի օպերատորներից յուրաքանչյուրում, օրինակ՝ *սկզբնարժեքավորում* օպերատորի մեջ: Ստորակետ , օպերատորը հրամանների տարանջատիչ է, և այն օգտագործվում է, որպեսզի տարանջատի մի քանի հրամաններ այնտեղ, որտեղ պետք է գրվի մեկ հրաման: Օրինակ՝ դիցուք մենք ուզում ենք օգտագործել երկու փոփոխականներ մեր ցիկլում: Այդ դեպքում կգրենք այսպես.

```
for ( n=0, m=100 ; n!=m ; n++, m-- )
{
    // մեր ծրագիրը
}
```

Այս ցիկլը կկատարվի 50 անգամ, եթե ցիկլի *մարմնում* ոչ `n` – ը, ոչ էլ `m` – ն չփոխվեն.

```
for ( n=0, m=100 ; n!=m ; n++, m-- )
```

 Սկզբնարժեքավորում Պայման Փոփոխություն

`n` – ը սկզբնարժեքավորվում է `0`, իսկ `m` – ն՝ `100`, և *պայմանը* (`n!=m`) է (`n` – ը հավասար չէ `m` – ի) է: Քանի որ ցիկլի ամեն քայլում `n` – ը մեծացվում է մեկով, իսկ `m` – ն՝ փոքրացվում, ապա ցիկլի *պայմանը* կդառնա `false` 50 քայլ հետո, երբ `n` – ը և `m` – ն հավասար լինեն `50` – ի:

Անցման հրամաններ (Jumps)

`break` հրամանը

Օգտագործելով `break` հրամանը, կարող ենք լքել ցիկլը, եթե նույնիսկ դրա ավարտի պայմանը չի կատարվել: Այս հրամանը կարող է օգտագործվել անվերջ ցիկլերից դուրս գալու համար, կամ հարկադրական եղանակով ցանկացած ցիկլ լքելու նպատակով: Օրինակ՝ հետևյալ ծրագրում ցիկլը կավարտվի `3` թվին հասնելիս.

```
#include <iostream>
using namespace std;

int main ()
{
```

```

int n;
for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
        cout << "cikl@ kangnecvec!";
        break;
    }
}
return 0;
}

```

10, 9, 8, 7, 6, 5, 4, 3, cikl@ kangnecvec!

continue հրամանը

continue հրամանը ստիպում է ծրագրին բաց թողնել ցիկլի մնացած մասը՝ սկսած այդ հրամանից: Այսինքն՝ երբ կանչվում է **continue** հրամանը, ցիկլի մարմինը համարվում է կատարված: Այս հրամանը ավելի լավ հասկանալու համար բերենք օրինակ: Հետևյալ օրինակում մենք շրջանցելու ենք 5 թիվը:

```

#include <iostream>
using namespace std;

int main ()
{
    for (int n=10; n>0; n--) {
        if (n==5) continue;
        cout << n << ", ";
    }
    cout << "VERJ!";
    return 0;
}

```

10, 9, 8, 7, 6, 4, 3, 2, 1, VERJ!

goto հրամանը

Այս հրամանը թույլ է տալիս կատարել անցում ծրագրի մի կետից մյուսին: Անցման կետը որոշվում է նշիչի միջոցով, որը նաև փոխանցում ենք **goto** հրամանին որպես արգումենտ: Նշիչն իրենից ներկայացնում է իդենտիֆիկատոր, որին հաջորդում է : սիմվոլը:

Օրինակ.

```

#include <iostream>
using namespace std;

int main ()
{
    int n=10;
loop:
    cout << n << ", ";
    n--;
    if (n>0) goto loop;
    cout << "VERJ!";
    return 0;
}

```

}
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, VERJ!

Շնորհության ստրուկտուրա՝ switch

switch հրամանի ֆունկցիան է համեմատել տրված արտահայտությունը այլ նախօրոք որոշված, հաստատուն արժեքների հետ: Սա մոտավորապես նույն բանն է, ինչ մենք արել էինք **if** - ի և **else if** - ի հետ: **switch** – ի տեսքը հետևյալն է.

```
switch ( արտահայտություն ) {
    case հաստատուն1:
        հրամանների_բլոկ_1
        break;
    case հաստատուն2:
        հրամանների_բլոկ_2
        break;
    .
    .
    .
    default:
        լռության_հրամանների_բլոկ
}
```

Սա աշխատում է հետևյալ կերպ. սկզբից **switch** - ը գնահատում է **արտահայտությունը** և ստուգում, թե արդյոք այն համարժեք է **հաստատուն1** - ին: Եթե դա իրոք այդպես է, ապա կատարվում է **հրամանների_բլոկ_1** – ը՝ մինչև **break** - ին հանդիպելը, որից հետո «թռչում է» **switch** - ի վերջ: Եթե **արտահայտությունը** համարժեք չէ **հաստատուն1** - ին, ապա ստուգվում է **արտահայտության** համարժեքությունը **հաստատուն2** - ին: Եթե համարժեք են, ապա կկատարվի **հրամանների_բլոկ_2** – ը՝ մինչև **break** - ին հանդիպելը: Եվ վերջապես, եթե **արտահայտությունը** չհամընկնի որևէ հաստատունի հետ (դուք կարող եք գրել այնքան **case** նախադասություններ, որքան կցանկանաք), ապա կկատարվի **լռության_հրամանների_բլոկը**, եթե այն ներկա է: Հատուկ նշենք նաև, որ լռության հրամանների բլոկը կարելի է չգրել: Այդ դեպքում եթե տրված արժեքին համապատասխան դեպք չգտնվի, ապա **switch** հրամանը ոչինչ չի կատարի:

Ստորև բերված երկու ծրագրերն իրար համարժեք են.

switch - ի օրինակ**if-else** - ով համարժեք օրինակ

<pre>switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }</pre>	<pre>if (x == 1) { cout << "x is 1"; } else { if (x == 2) { cout << "x is 2"; } else { cout << "value of x unknown"; } }</pre>
--	--

Ուշադրություն դարձրեք նաև այն հանգամանքին, որ յուրաքանչյուր *հրամանների_բլոկից* հետո գրված է **break**: Եթե, օրինակ, վերացնենք *հրամանների_բլոկ_1* – ից հետո գրված **break** - ը, ապա ծրագիրը, հասնելով *հրամանների_բլոկ_1* – ի վերջին, չի անցնի **switch** - ի վերջին, այլ կշարունակի կատարել իրենից ներքև գրված ծրագիրը՝ մինչև **break** - ին կամ **switch** - ի վերջին հանդիպելը: Դա է պատճառը, որ հարկ չկա սահմանափակելու յուրաքանչյուր **case** - ում գրված *հրամանների_բլոկը* ձևավոր փակագծերով, և, ավելին, դա հնարավորություն է տալիս օգտագործել միևնույն *հրամանների_բլոկը* տարբեր դեպքերի համար: Օրինակ.

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x-@ 1 e, 2 kam 3";
    break;
  default:
    cout << "x-@ voch 1 e, voch 2, voch e1 3";
}
```

Հիշեք նաև, որ **switch** – ը կարող է օգտագործվել միայն տրված *արտահայտությունը* նշված հաստատունների հետ համեմատելու համար: Օրինակ՝ չենք կարող գրել **case (n*2)**: Եթե ուզում եք տրված *արտահայտությունը* համեմատել փոփոխականների կամ ինչ-որ տիրույթների հետ, ապա օգտագործեք **if-else** հրամանները:

Բաժին 2.2

Ֆունկցիաներ (I)

Օգտագործելով ֆունկցիաներ՝ կարող ենք ծրագիրը բաժանել առանձին հատվածների՝ հնարավորություն ստանալով արդեն պատրաստ ծրագրի մասերն օգտագործել այլ ծրագրերի մեջ: Ֆունկցիաները **կառուցվածքային ծրագրավորման** (structured programming) հիմնական գաղափարներից են:

Ֆունկցիան հրամանների հաջորդականություն է, որոնք աշխատեցվում են, երբ ֆունկցիան կանչվում է ծրագրի որևէ մասից: Ֆունկցիայի գրելաձևը հետևյալն է՝

տիպ անուն (արգումենտ1, արգումենտ2, ...) մարմին

որտեղ

- ***տիպը*** ֆունկցիայի վերադարձվող արժեքի տիպն է,
- ***անունը*** այն անունն է, որը պետք է օգտագործենք ֆունկցիան կանչելիս,
- ***արգումենտները*** այն արժեքներն են, որոնք փոխանցում ենք ֆունկցիային: Կարելի է ունենալ այնքան արգումենտ, որքան անհրաժեշտ է: Յուրաքանչյուր արգումենտ կազմված է տիպից և նույնարկիչից, որով այն պիտի տարբերվի մյուսներից (օրինակ՝ **int x**). սա նման է փոփոխականների հայտարարման: Նշված արգումենտը ֆունկցիայի մեջ իրեն «պահում է» ճիշտ այնպես, ինչպես ցանկացած փոփոխական: Արգումենտները իրարից բաժանված են ստորակետերով:
- ***մարմինը*** հրամաններն են, որոնք պիտի կատարի ֆունկցիան: Ֆունկցիայի մարմինը կարող է կազմված լինել միակ հրամանից կամ հրամանների բլոկից: Վերջին դեպքում դրանք վերցվում են փակագծերի մեջ՝ **{ }**:

Ահա մի ֆունկցիայի օրինակ.

```
// funkciiyi orinak
#include <iostream>
using namespace std;

int gumarum(int a, int b)
{
    int r;
    r = a + b;
    return (r);
}

int main ()
{
    int z;
    z = gumarum(5,3);
    cout << "Ardyunq@ exav " << z;
    return 0;
}
```

Ardyunq@ exav 8

Որպեսզի սկսենք քննարկել այս ծրագիրը, հիշենք, որ, ինչպես գրված էր այս ձեռնարկի նախորդ գլուխներում, **C++** - ում ծրագրի աշխատանքը սկսվում է **main** ֆունկցիայի կանչով: Այդտեղից էլ սկսենք քննարկել:

Ինչպես տեսնում ենք, **main** ֆունկցիան սկսվում է **int** տիպի **z** փոփոխականի հայտարարմամբ: Դրանից անմիջապես հետո կատարված է դիմում **gumarum** ֆունկցիային: Եթե ուշադրություն դարձնենք ֆունկցիայի կանչին, ապա կարող ենք հեշտությամբ կապ գտնել ֆունկցիայի հայտարարման և նրան դիմելու ձևի միջև՝

```
int gumarum ( int a, int b )  
z = gumarum ( 5 , 3 );
```

main ֆունկցիայի միջից կանչում ենք **gumarum** ֆունկցիան՝ նրան փոխանցելով **5** և **3** արժեքները, որոնք համապատասխանում են **int a** և **int b** պարամետրերին՝ հայտարարված **gumarum** ֆունկցիայում:

Այն ժամանակ, երբ ֆունկցիան կանչվում է **main** - ի միջից, ծրագրի աշխատանքի կառավարումը **main** - ից անցնում է **gumarum** ֆունկցիային: Կանչի ժամանակ ֆունկցիային փոխանցված երկու պարամետրերը՝ (**5** և **3**) արտագրվում են **int a** և **int b** փոփոխականների մեջ, որոնք **gumarum** ֆունկցիայի լոկալ փոփոխականներ են:

gumarum ֆունկցիան հայտարարում է մի նոր փոփոխական (**int r**;) և **r = a + b**; արտահայտության միջոցով **r** - ին վերագրում **a** և **b** թվերի գումարման արդյունքը: Տրամաբանական է ենթադրել, որ արդյունքում կստացվի **8**:

Կողի հետևյալ տողը՝

```
return ( r );
```

ֆունկցիայի աշխատանքը հասցնում է իր տրամաբանական ավարտին և ծրագրի աշխատանքի կառավարումը հետ է փոխանցում **main** ֆունկցիային (**main** - ը շարունակում է իր աշխատանքը ճիշտ այն տողից, որտեղից ընդհատվել էր): Բայց բացի դրանից, **return** - ը կանչելիս նրան տրվել էր փոփոխական՝ **r**, որն այդ ժամանակ իր մեջ պահում էր **8** արժեքը. այսպիսով ֆունկցիան վերադարձնում է այդ արժեքը:

```
int gumarum ( int a, int b )  
↓ 8  
z = gumarum ( 5 , 3 );
```

Ինչպես մաթեմատիկայում **f(3)** գրելով հասկանում ենք այն արժեքը, որը ստանում է ֆունկցիան նրա մեջ **x** - ի փոխարեն **3** տեղադրելիս, այնպես էլ այստեղ ֆունկցիայի կանչը (**gumarum(5, 3)**) «փոխարինվում է» այն արժեքով, որ վերադարձնում է ֆունկցիան. կոնկրետ դեպքում դա **8** ամբողջ թիվն է:

main ֆունկցիայի հաջորդ տողը, ինչպես երևի գուշակեցիք, էկրանի վրա գրում է գումարման արդյունքը՝


```
cout << "Ardyung@ exav " << z;
```

Փոփոխականների տեսանելիության տիրույթ [կրկնություն]

Վերհիշենք, որ ֆունկցիայում հայտարարված փոփոխականների տեսանելիության տիրույթը սահմանափակված է միայն այդ ֆունկցիայով, և հետևաբար այդ փոփոխականները ֆունկցիայից դուրս օգտագործվել չեն կարող: Նախորդ օրինակում **main** ֆունկցիայի միջից չենք կարող անմիջականորեն դիմել **a**, **b** և **r** փոփոխականներին, քանզի նրանք հայտարարված են **gumarum** ֆունկցիայում: Իսկ **gumarum** ֆունկցիայից չենք կարող դիմել **main** - ում հայտարարված **z** փոփոխականին:

Բայց միշտ կարող ենք հայտարարել գլոբալ փոփոխականներ, որոնք հասանելի կլինեն կոդի ցանկացած մասից: Դրա համար փոփոխականը պետք է հայտարարել բոլոր ֆունկցիաներից դուրս:

Ֆունկցիաների վերաբերյալ մեկ այլ օրինակ.

```
// funksiayi orinak
#include <iostream>
using namespace std;

int hanum(int a, int b)
{
    int r;
    r = a - b;
    return (r);
}

int main ()
{
    int x=5, y=3, z;
    z = hanum(7,2);
    cout << "Arajin ardyung@: " << z << '\n';
    cout << "Yerkrord ardyung@: " << hanum(7,2) << '\n';
    cout << "Yerrord ardyung@: " << hanum(x,y) << '\n';
    z = 4 + hanum(x,y);
    cout << "Chorrord ardyung@: " << z << '\n';
    return 0;
}
```

```
Arajin ardyung@: 5
Yerkrord ardyung@: 5
Yerrord ardyung@: 2
Chorrord ardyung@: 6
```

Այս անգամ ստեղծեցինք **hanum** ֆունկցիան, որը առաջին արգումենտից հանում է մյուսը և վերադարձնում արդյունքը:

Այս օրինակում **main** ֆունկցիայից **hanum** ֆունկցիան մի քանի անգամ ենք կանչել՝ ամեն անգամ մի նոր ձևով, այնպես, որ պարզ դառնան ֆունկցիայի կանչի հնարավոր տարբերակները:

Քննարկենք օրինակը՝ տող առ տող.

```
z = hanum(7, 2);  
cout << "Arajin ardyunq@: " << z;
```

Եթե ֆունկցիայի կանչը փոխարինենք նրա վերադարձրած արժեքով (որը, ի դեպ, 5 է), կստացվի՝

```
z = 5;  
cout << "Arajin ardyunq@: " << z;
```

Նույնը՝ հաջորդ տողի վերաբերյալ.

```
cout << "Yerkrord ardyunq@: " << hanum(7, 2);
```

hanum(7, 2) - ը փոխարինելով արդյունքով՝ 5 կստանանք.

```
cout << "Yerkrord ardyunq@: " << 5;
```

Նկատենք, որ

```
cout << "Yerord ardyunq@: " << hanum(x, y);
```

դեպքում ֆունկցիայի արգումենտները փոփոխականներ են, այլ ոչ թե հաստատուններ: Այս դեպքում ֆունկցիան կանչելիս փոփոխականների արժեքներն են փոխանցվում ֆունկցիային (կարող ենք համարել, որ փոփոխականները փոխարինվում են իրենց արժեքներով՝ ինչպես ֆունկցիան է փոխարինվում իր արժեքով):

Հաջորդ դեպքը մոտավորապես նույնն է ինչ որ նախորդը.

```
z = 4 + hanum(x, y);
```

սկզբից հաշվվում է ֆունկցիայի արժեքը, այնուհետև նրան ավելացվում է 4, և արդյունքը գրվում է z փոփոխականի մեջ: Արդյունքը ճիշտ նույնը կլինի, եթե գրեինք.

```
z = hanum(x, y) + 4;
```

Ֆունկցիաներ՝ առանց տիպի: *void* - ի օգտագործումը

Ինչպես հիշում եք՝ ֆունկցիայի գրելաձևը հետևյալն է՝

տիպ *անուն* (*արգումենտ1*, *արգումենտ2*, ...) *մարմին*

Այն սկսվում է *տիպով*, բայց ինչպե՞ս վարվել, եթե չենք ուզում, որ ֆունկցիան որևէ արժեք վերադարձնի:

Ենթադրենք՝ ուզում ենք գրել ֆունկցիա, որն էկրանի վրա որևէ տեքստ է տպում: Այդ դեպքում որևէ արժեք վերադարձնելու կարիք չունենք, ինչպես նաև կարիք

չունենք նրան արգումենտ փոխանցելու: Այս և նմանատիպ պատճառներով ստեղծվել է **void** տիպը: Դիտարկենք հետևյալ օրինակը`

```
// void funkciayi orinak
#include <iostream>
using namespace std;

void inchvorfunkcia(void)
{
    cout << "Yes funkcia em!";
}

int main()
{
    inchvorfunkcia();
    return 0;
}
```

Yes funkcia em!

Ֆունկցիայի արգումենտներում կարելի է **void** - ը չգրել. ամեն դեպքում թարգմանիչը <<հասկանում է>>, որ ֆունկցիային արգումենտներ չեն փոխանցվելու:

Հիշենք, որ նույնիսկ եթե ֆունկցիան արքումենտներ չի ընդունում, փակագծերը () պետք է միշտ դնել`

inchvorfunkcia();

Դրանով պարզ է դառնում, որ նշվածը ֆունկցիայի կանչ է, այլ ոչ թե փոփոխական կամ մեկ այլ բան:

Բաժին 2.3

Ֆունկցիաներ (II)

Արգումենտների փոխանցումը արժեքով (by value) և հասցեով (by reference)

Մինչը հիմա մեր քննարկած բոլոր օրինակներում արգումենտը ֆունկցիային փոխանցվում էր արժեքով: Սա նշանակում է, որ երբ կանչվում էր արգումենտներով ֆունկցիա, մենք փոխանցում էինք փոփոխականների արժեքները, այլ ոչ թե փոփոխականները: Դիցուք կանչում ենք նախորդ գլխում քննարկված գումարման ֆունկցիան հետևյալ ձևով՝

```
int x=5, y=3, z;  
z = gumarum ( x , y );
```

Այս դեպքում ֆունկցիան կանչելիս փոխանցեցինք ոչ թե **x** և **y** փոփոխականները, այլ համապատասխանաբար **5** և **3** թվերը:

```
int gumarum ( int a, int b )  
                ↑5   ↑3  
z = gumarum ( x , y );
```

Նման ձևով ֆունկցիան կանչելիս **a** և **b** փոփոխականներում կգրվեն **5** և **3** թվերը, և ֆունկցիայում **a** կամ **b** փոփոխականի փոփոխությունը ազդեցություն չի ունենա **x** և **y** փոփոխականների արժեքների վրա. այսինքն՝ փոխանցվել են ոչ թե **x** և **y** փոփոխականները, այլ նրանց արժեքները:

Սակայն որոշ դեպքերում կարող է հարկ լինի փոփոխել արտաքին փոփոխականները, որոնք փոխանցվում են ֆունկցիային՝ որպես արգումենտ: Այս դեպքում պետք է փոփոխականները փոխանցել հասցեով՝ ինչպես արված է հաջորդ օրինակում:

```
// argumentneri poxancum` hasceov  
#include <iostream>  
using namespace std;  
  
void krknapatik (int& a, int& b, int& c)  
{  
    a*=2;  
    b*=2;  
    c*=2;  
}  
  
int main ()  
{  
    int x=1, y=3, z=7;  
    krknapatik (x, y, z);  
    cout << "x=" << x << ", y=" << y << ", z=" << z;  
    return 0;  
}
```

x=2, y=6, z=14

Առաջինը, որ ուշադրություն է գրավում, **krknapatik** ֆունկցիայի հայտարարման մեջ տարօրինակ **&** սիմվոլների առկայությունն է. դրանք օգտագործվում են՝ նշելու, որ արգումենտները փոխանցվում են հասցեով, այլ ոչ թե արժեքով՝ ինչպես սովորաբար:

Երբ փոփոխականը փոխանցում ենք հասցեով, փոխանցվում է ոչ թե փոփոխականի արժեքը, այլ հենց փոփոխականը: Այսինքն՝ ֆունկցիայում՝ նրան փոխանցված փոփոխականի ցանկացած փոփոխությունը կազդի նաև սկզբնական փոփոխականի վրա:

```
void krknapatik ( int& a, int& b, int& c )
                ↑↑x   ↑↑y   ↑↑z
krknapatik ( x , y , z )
```

Այլ կերպ ասած՝ որպես **a**, **b** և **c** արգումենտներ ֆունկցիային փոխանցել ենք **x**, **y** և **z** փոփոխականները, և եթե ֆունկցիայի մեջ **a** - ի արժեքը փոխենք, ապա դրսում կփոխվի նաև **x** - ի արժեքը: **b** - ի վրա կատարված ցանկացած փոփոխություն կրերի **y** - ի փոփոխության. նույնը՝ **c** - ի և **z** - ի դեպքում:

Հենց այս պատճառով **x**, **y** և **z** փոփոխականների արժեքները կրկնապատկվեցին:

Եթե

```
void krknapatik(int& a, int& b, int& c)
```

ֆունկցիան հայտարարելիս գրեինք

```
void krknapatik(int a, int b, int c)
```

(առանց **&** նշանների) , ապա արգումենտները փոխանցված կլինեին արժեքով, այլ ոչ թե հասցեով: Այդ դեպքում **a**, **b** կամ **c** փոփոխականները փոփոխելիս **x**, **y** և **z** - ը կմնային անփոփոխ:

Արգումենտները հասցեով փոխանցելը ֆունկցիաներին հնարավորություն է տալիս վերադարձնել մի քանի արժեքներ միանգամից: Հաջորդ օրինակը վերադարձնում է տրված թվի նախորդ և հաջորդ թվերը՝ օգտագործելով արգումենտների փոխանցումը հասցեով:

```
// 1-ic avel veradarzvox arjeqner
#include <iostream>
using namespace std;

void naxordhajord (int x, int& prev, int& next)
{
    prev = x-1;
    next = x+1;
}

int main ()
{
    int x=100, y, z;
    naxordhajord (x, y, z);
    cout << "Naxord=" << y << ", Hajord=" << z;
```

```
return 0;
}
Naxord=99, Hajord=101
```

Արգումենտների լռության (Default) արժեքներ

Ֆունկցիա հայտարարելիս կարող ենք յուրաքանչյուր արգումենտի համար նշել լռության արժեք: Այդ արժեքները կօգտագործվեն, եթե ֆունկցիան կանչելիս այդ արգումենտները դատարկ թողնվեն: Դա կատարելու համար պարզապես ֆունկցիան հայտարարելիս արգումենտները հավասարեցնում ենք անհրաժեշտ արժեքներին: Այժմ եթե ֆունկցիան կանչելիս այդ արգումենտները դատարկ թողնենք, դրանք կփոխարինվեն լռության արժեքներով, իսկ եթե դատարկ չթողնենք, ապա լռության արժեքները կանտեսվեն, և կգործեն ֆունկցիային արգումենտներ փոխանցելու սովորական օրենքները: Օրինակ.

```
// lruty an arjeqner
#include <iostream>
using namespace std;

int bajanel (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}

int main ()
{
    cout << bajanel (12);
    cout << endl;
    cout << bajanel (20,4);
    return 0;
}

6
5
```

Ինչպես տեսնում եք, ծրագրի մարմնում կա երկու կանչ **bajanel** ֆունկցիային: Առաջինում՝

```
bajanel(12)
```

նշեցինք միայն մի արգումենտ, և մյուսը փոխարինվեց իր լռության արժեքով՝ **2** - ով: Դա կատարվում է, որովհետև ֆունկցիայի նկարագրման մեջ գրված է **int b=2**, և ֆունկցիայի կանչի մեջ պակասող արգումենտը փոխարինվում է նրա համար նախապես նշված արժեքով: Արդյունքում ստացվում է **6**:

Մյուս կանչի մեջ՝

```
bajanel(20,4)
```

բոլոր արգումենտները «տեղում են». լրության արժեքը պարզապես անտեսվում է, և գործում են արգումենտի փոխանցման ստանդարտ կանոնները: Արդյունքում ստացվում է 5` (20 / 4):

Ֆունկցիաների գերբեռնում (Overloading)

Երկու ֆունկցիաներ կարող են ունենալ միևնույն անունը, եթե նրանց արգումենտների ցուցակները տարբեր են. այսինքն` մենք կարող ենք երկու ֆունկցիայի տալ միևնույն անունը, եթե նրանք ընդունում են տարբեր քանակությամբ արգումենտներ, կամ տարբերվում են ընդունվող արգումենտների տիպերը: Օրինակ`

```
// gerbernva& funkcija
#include <iostream>
using namespace std;

int bajanel(int a, int b)
{
    return (a / b);
}

float bajanel(float a, float b)
{
    return (a / b);
}

int main()
{
    int x=5,y=2;
    float n= 5.0,m=2.0;
    cout <<bajanel(x, y);
    cout << "\n";
    cout << bajanel(n, m);
    cout << "\n";
    return 0;
}
```

```
2
2.5
```

Այստեղ ունենք նույն անունով երկու ֆունկցիա, բայց նրանցից մեկն ընդունում է **int** տիպի երկու փոփոխական, իսկ մյուսը` **float** տիպի երկու փոփոխական: Թարգմանիչը գիտի` որ դեպքում որ ֆունկցիան կանչել: Երբ այդ ֆունկցիան կանչում ենք երկու **int** տիպի պարամետրերով, ապա թարգմանիչի համար պարզ է դառնում, որ պետք է կանչել երկու **int** տիպի արգումենտ ունեցող ֆունկցիան: Իսկ **float** - ի դեպքում կանչվում է **float** տիպի արգումենտներ ընդունող ֆունկցիան:

Կոնկրետ այս դեպքում մեր օրինակում երկու **bajanel** ֆունկցիաները կատարում են նույն գործողությունները` առաջին թիվը բաժանում են երկրորդի վրա, սակայն դրանք կարող են կատարել բոլորովին տարբեր գործողություններ:

inline (տողամիջյան) ֆունկցիաներ

Ֆունկցիայի հայտարարումից առաջ նշելով **inline** հրամանը՝ ֆունկցիան դարձնում ենք տողամիջյան: Այս ֆունկցիաների ամեն կանչը փոխարինվում է ֆունկցիայի մարմնով: Սովորաբար **inline** հայտարարում են միայն կարճ ֆունկցիաները: Այս դեպքում ֆունկցիայի կանչի վրա ժամանակ չի ծախսվում:

Գրելաձևը հետևյալն է.

```
inline տիպ անուն ( արգումենտներ ... ) { հրամաններ ... }
```

Իսկ կանչում ենք ճիշտ այնպես, ինչպես ցանկացած սովորական ֆունկցիա: Հարկավոր չէ **inline** հրամանը գրել ֆունկցիայի յուրաքանչյուր կանչի մեջ. բավական է այն գրել միայն ֆունկցիայի հայտարարման մեջ:

Բեկուրսիա (Recursivity)

Բեկուրսիան այն պրոցեսն է, երբ ֆունկցիան կանչում է ինքն իրեն: Դա շատ օգտակար է, օրինակ, թվի ֆակտորիալը հաշվելիս: Ինչպես գիտենք n թվի ֆակտորիալը հետևյալ թիվն է՝

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

ավելի կոնկրետ՝ 5 թվի ֆակտորիալը կլինի

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

և դա կարելի է անել հետևյալ բեկուրսիվ ֆունկցիայի միջոցով՝

```
// factorial hashvox &ragir
#include <iostream>
using namespace std;

long factorial(long a) {
    if (a >1)
        return (a * factorial(a-1));
    else return(1);
}
int main() {
    long l;
    cout << "Tiv gri: ";
    cin >> l;
    cout <<l<<"! = " << factorial(l);
    return 0;
}
```

```
Tiv gri: 9
9! = 362880
```


Ուշադրություն դարձնենք, թե ինչպես է **factorial** ֆունկցիան կանչում ինքն իրեն, բայց միայն այն դեպքում երբ արգումենտը **1** - ից մեծ է. այլապես ծրագիրը կընկներ անվերջ ռեկուրսիվ ցիկլի մեջ հասնելով **0** – ին, այնուհետև կշարունակեր բազմապատկել բացասական թվերով՝ վերջ ի վերջո առաջացնելով սխալ:

Այս ֆունկցիան ունի որոշակի սահմանափակումներ. այն ֆակտորիալը պահում է **long** - ի մեջ, իսկ այնտեղ դժվար թե տեղավորվեն **12!** - ից մեծ թվեր:

Ֆունկցիայի նախատիպ (Prototype)

Մինչև հիմա մենք ֆունկցիաները հայտարարում էինք մինչև նրանց առաջին կանչը, որը մեզ մոտ կատարվում էր **main** ֆունկցիայում. **main** - ը հայտարարում էինք վերջում: Եթե ֆունկցիաների նախորդ օրինակներից մեկում **main** - ը բերենք մյուս ֆունկցիաներից (որոնք կանչվում են **main** - ից) առաջ, ապա հավանաբար կստանանք սխալ: Պատճառը այն է, որ ցանկացած ֆունկցիա պետք է հայտարարված լինի մինչև կանչելը (հենց այդպես էր արված նախորդ օրինակներում):

Բայց կա այլընտրանքային եղանակ, որը սահմանափակում չի դնում ֆունկցիաների հերթականության վրա: Այդ եղանակը ֆունկցիայի նախատիպը տալն է: Այս ձևով կարող ենք որևէ ֆունկցիա օգտագործելուց առաջ տալ նրա նախատիպը, որը կազմված է միայն ֆունկցիայի վերադարձվող արժեքի տիպից, անունից և արգումենտներից (ֆունկցիայի մարմինը չկա): Այս դեպքում թարգմանիչը արդեն գիտի, որ կա այդպիսի ֆունկցիա, գիտի նրա ընդունած արգումենտները և վերադարձվող արժեքի տիպը: Սրանից հետո ֆունկցիան՝ իր մարմնով, կարող ենք գրել ցանկացած տեղ:

Գրելաձևը՝

տիպ անուն (**արգումենտի_տիպ_1**, **արգումենտի_տիպ_2**, ...);

Սա լիովին համընկնում է ֆունկցիայի հայտարարման վերնագրի հետ՝ հետևյալ բացառություններով.

- այն չի պարունակում ֆունկցիայի մարմինը. այսինքն, բացակայում են { } նշանները և նրանց մեջ գրվող հրամանները:
- այն վերջանում է կետ-ստորակետով՝ ; :
- բավարար է նշել միայն արգումենտների տիպերը: Մակայն խորհուրդ է տրվում գրել նաև նրանց անունները՝ ինչպես սովորական ֆունկցիայի հայտարարման մեջ:

Օրինակ.

```
// naxatip
#include <iostream>
using namespace std;

void kent(int a);
void zuyg(int a);
int main()
{
```

```

int i;
do {
    cout<< "Grir voreve tiv (0 elqi hamar): ";
    cin >> i;
    kent(i);
} while (i!=0);
return 0;
}

void kent(int a)
{
    if ((a%2)!=0) cout << "Tiv@ kent e.\n";
    else zuyg(a);
}

void zuyg(int a)
{
    if ((a%2)==0) cout << "Tiv@ zuyg e.\n";
    else kent(a);
}

```

```

Grir voreve tiv (0 elqi hamar): 9
Tiv@ kent e.
Type a number (0 to exit): 6
Tiv@ zuyg e.
Type a number (0 to exit): 1030
Tiv@ zuyg e.
Type a number (0 to exit): 0
Tiv@ zuyg e.

```

Այս օրինակը, իսկապես, արդյունավետ օրինակ չէ, սակայն այն շատ լավ ցույց է տալիս ֆունկցիաների նախատիպերի կիրառումն ու անհրաժեշտությունը. այս օրինակում ֆունկցիաներից գոնե մեկը պետք է ունենա նախատիպ:

Հետևյալ երկու տողերը **kent** և **zuyg** ֆունկցիաների նախատիպերն են

```

void kent(int a);
void zuyg(int a);

```

Սա հնարավորություն է տալիս օգտագործել այս ֆունկցիաները՝ մինչև հայտարարումը:

Այս օրինակում **kent** և **zuyg** ֆունկցիաներից մեկը անպայման պետք է ունենա նախարիպ, քանի որ **kent** ֆունկցիան ունի **zuyg** ֆունկցիայի կանչ և հակառակը: Եթե **zuyg** - ը առաջինը գրեինք, իսկ **kent** – ը նրանից հետո, ապա կառաջանար սխալ, քանի որ **kent** - ը կանչվում է մինչև հայտարարվելը՝ **zuyg** - ի միջից:

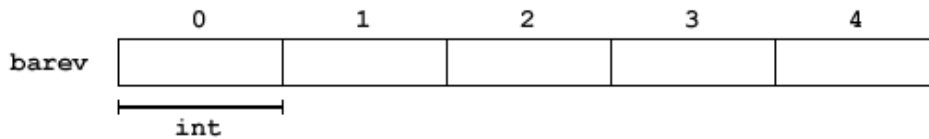
Խորհուրդ է տրվում նախատիպեր սահմանել բոլոր ֆունկցիաների համար, քանի որ, երբ բոլոր ֆունկցիաների նախատիպերը գրված են միևնույն տեղում, ավելի հեշտ է գտնել մեզ անհրաժեշտ ֆունկցիան: Ֆունկցիաների նախատիպերը սովորաբար գրում են առանձին՝ **վերնագիր** (header) ֆայլերում, այնուհետև դրանք կցում ծրագրին:

Բաժին 3.1

Չանգվածներ (Arrays)

Չանգվածն իրենից ներկայացնում է նույն տիպի տարրերի (փոփոխականների) խումբ, որոնք հաջորդաբար դասավորված են հիշողության մեջ, և որոնց կարող ենք դիմել՝ զանգվածի անվանը կցելով համապատասխան տարրի համարը (ինդեքսը):

Սա նշանակում է, որ մենք կարող ենք պահել 5 `int` տիպի փոփոխականներ և ստիպված չենք լինի 5 – ի համար էլ գրել հայտարարությունները և դրանց տալ 5 տարբեր անուններ: Օրինակ՝ 5 հատ `int` տիպի փոփոխականներ պարունակող `barev` անունով զանգվածը կարող է ներկայացվել հետևյալ կերպ.



որտեղ յուրաքանչյուր դասարկ ուղղանկյունն իրենից ներկայացնում է զանգվածի տարր, որը, մեր դեպքում, `int` տիպի է: Այս էլեմենտները համարակալված են 0 - 4, քանի որ զանգվածներում առաջին տարրի համարը միշտ 0 է՝ անկախ զանգվածի երկարությունից:

Ինչպես և բոլոր մյուս փոփոխականները, զանգվածը պետք է հայտարարված լինի մինչև նրա օգտագործումը: `C++` - ում զանգվածները հայտարարվում են հետևյալ կերպ՝

տիպ անուն [տարրերի քանակ];

որտեղ *տիպը* զանգվածի տարրերի տիպն է, *անունը* զանգվածի անունն է, իսկ *տարրերի քանակը*՝ զանգվածի տարրերի քանակը: Այսպիսով, `int` տիպի 5 տարր պարունակող `barev` անունով զանգված հայտարարելու համար պետք է գրել

```
int barev [5];
```

ՈՒՇԱԴՐՈՒԹՅՈՒՆ՝ զանգվածի հայտարարման ժամանակ *տարրերի քանակը* պետք է լինի հաստատուն թիվ, քանի որ զանգվածները տրված չափով ստատիկ հիշողության բլոկներ են, և թարգմանիչը պետք է կարողանա որոշել, թե ինչքան հիշողություն պետք է անջատի զանգվածի համար, մինչև որևէ հրամանի կատարումը:

Չանգվածների սկզբնաբժեքավորումը (Initialization of Arrays)

Լոկալ տեսանելիության տիրույթում (ֆունկցիայում) զանգված հայտարարելիս այն ավտոմատ չի սկզբնաբժեքավորվելու, և դրա պարունակությունը բացահայտված չի լինի, քանի դեռ դրան վերագրվեն ինչ-որ արժեքներ:

Իսկ գլոբալ տեսանելիության տիրույթում (բոլոր ֆունկցիաներից դուրս) զանգված հայտարարելիս, դրա պարունակությունը ավտոմատ սկզբնաբաժանված է գրոներով: Այսպիսով, եթե մենք գլոբալ տեսանելիության տիրույթում հայտարարենք

```
int barev [5];
```

այս **barev** զանգվածի յուրաքանչյուր տարրում գրված կլինի **0**

	0	1	2	3	4
barev	0	0	0	0	0

Չանգված հայտարարելիս նաև ունենք հնարավորություն նրա յուրաքանչյուր տարրի որևէ արժեք վերագրել՝ օգտագործելով ձևավոր փոկագծեր: Օրինակ՝

```
int barev [5] = { 16, 2, 77, 40, 12071 };
```

այս հայտարարությամբ կստեղծվի հետևյալ զանգվածը.

	0	1	2	3	4
barev	16	2	77	40	12071

Ձևավոր փակագծերի մեջ գրված տարրերի քանակը պետք է համընկնի զանգվածի *տարրերի քանակին*: Մեր դեպքում զանգվածի *տարրերի քանակը* 5 է, այդ է պատճառը, որ ձևավոր փակագծերի մեջ գրված տարրերի քանակը նա 5 է:

Կարող ենք նաև բաց թողնել զանգվածի *տարրերի քանակը*՝ թողնելով այդ գործը թարգմանչին: Այդ դեպքում որպես զանգվածի *տարրերի քանակ* կընդունվի ձևավոր փակագծերի մեջ գրված տարրերի քանակը.

```
int barev [] = { 16, 2, 77, 40, 12071 };
```

Չանգվածների տարրերին դիմումը (Access to the Values of an Array)

Չանգվածի տեսանելիության տիրույթի ցանկացած կետում կարող ենք կարդալ կամ փոփոխել տրված զանգվածի ցանկացած տարր: Տեսքը հետևյալն է.

```
զանգվածի անուն [ նամար ]
```

Նախորդ օրինակում բերված **barev** զանգվածի տարրերին կարող ենք դիմել հետևյալ համապատասխան անուններով.

	barev[0]	barev[1]	barev[2]	barev[3]	barev[4]
barev					

Օրինակ՝ **barev** զանգվածի 3 - րդ տարրում 75 գրելու համար, կարող ենք գրել

```
barev[2] = 75;
```

և, օրինակ, **a** փոփոխականին **barev** զանգվածի 3 – րդ տարրի արժեքը վերագրելու համար կգրենք

```
a = barev[2];
```

barev[2] արտահայտությունը համարժեք է ցանկացած **int** տիպի փոփոխականի:

Ուշադրություն դարձնենք նաև այն բանին, որ **barev** զանգվածի 3 - րդ տարրը **barev[2]** - ն է, առաջինը՝ **barev[0]** - ն, իսկ վերջինը՝ **barev[4]** - ր:

C++ - ում բավականին տարածված սխալ է զանգվածի՝ գոյություն չունեցող տարրին դիմելը, և քանի որ թարգմանիչը դա սխալ չի համարում, ապա ծրագրի աշխատանքի ժամանակ դա կարող է հանգեցնել ծրագրի սխալ արդյունքի: Այս խնդիրը ավելի մանրամասն կքննարկենք հետագա դասերում, երբ անցնենք ցուցիչ տիպեր:

Տվյալ պահին կարևոր է տարբերել իրարից, քառակուսի փակագծերի՝ [], երկու՝ իրարից տարբեր իմաստները: Առաջին՝ նրանք օգտագործվում են զանգվածների հայտարարման ժամանակ՝ զանգվածի չափը որոշելու համար, և երկրորդ՝ օգտագործվում են տրված զանգվածի որևէ տարրի դիմելու համար.

```
int barev[5]; // nor zangvaci haitararutium (sksvum e tipi anunov)
barev[2] = 75; // zangvaci elementi dimum
```

Այլ գործողություններ զանգվածների հետ.

```
barev[0] = a;
barev[a] = 75;
b = barev[a+2];
barev[barev[a]] = barev[2] + 5;
```

```
// zangvaci orinak
#include <iostream>
using namespace std;

int barev [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
    for ( n=0 ; n<5 ; n++ )
    {
        result += barev[n];
    }
    cout << result;
    return 0;
}
```

12206

Բազմաչափ զանգվածներ (Multidimensional Arrays)

Բազմաչափ զանգվածները կարող են նկարագրվել որպես զանգվածների զանգվածներ: Ստորև բերված է երկչափ զանգված:

		0	1	2	3	4
matrix	0					
	1					
	2					

matrix - ը իրենից ներկայացնում է 3 - ը 5 - ի `int` տիպի երկչափ զանգված: Այսպիսի զանգվածի հայտարարման ձևը հետևյալն է.

```
int matrix [3][5];
```

և, օրինակ, 2 - ը տողի 4 - ը տարրին դիմելու համար պետք է օգտագործել հետևյալ արտահայտությունը՝

		0	1	2	3	4
matrix	0					
	1				●	
	2					

↓
matrix[1][3]

(հարկավոր է հիշել, որ զանգվածի ինդեքսները սկսվում են 0 - ից):

Բազմաչափ զանգվածները կարող են լինել ինչպես երկչափ, այնպես էլ եռաչափ, քառաչափ և այլն: Չնայած, սովորաբար, եռաչափ զանգվածներից մեծ զանգվածներ գրեթե չեն օգտագործվում: Պատկերացրեք միայն, թե ինչքան հիշողություն անհրաժեշտ կլինի հետևյալ զանգվածը պահելու համար.

```
char zangvac [100][365][24][60][60];
```

Հաշվի առնելով, որ `char` - ը զբաղեցնում է 1 բայթ՝ կստանանք ավելի քան 3 միլիարդ բայթ հիշողություն, այսինքն՝ մոտավորապես 3 ԳիգաԲայթ հիշողություն:

Բազմաչափ զանգվածները պարզապես ստեղծված են ծրագրավորողի հարմարավետության համար, քանի որ իրականում կարող ենք կատարել նույն գործը միաչափ զանգվածով.

```
int matrix [3][5]; //համարժեք է՝  
int matrix [15];  //(3 * 5 = 15)
```

Միակ տարբերություն այն է, որ թարգմանիչը մեզ համար հիշում է յուրաքանչյուր երևակայած տարածության չափը: Ստորև բերված է մի ծրագիր, որը գրված է երկու ձևով. մի դեպքում օգտագործելով միաչափ զանգված, մյուս դեպքում՝ երկչափ.

<pre>// miachap zangvac #include <iostream> using namespace std; #define WIDTH 5 #define HEIGHT 3 int matric [HEIGHT * WIDTH]; int n,m; int main () { for (n=0;n<HEIGHT;n++) { for (m=0;m<WIDTH;m++) { matric[n * WIDTH + m]=(n+1)*(m+1); } } return 0; }</pre>	<pre>// bazmachap zangvac #include <iostream> using namespace std; #define WIDTH 5 #define HEIGHT 3 int matric [HEIGHT][WIDTH]; int n,m; int main () { for (n=0;n<HEIGHT;n++) { for (m=0;m<WIDTH;m++) { matric[n][m]=(n+1)*(m+1); } } return 0; }</pre>
--	--

Այս ծրագրերից ոչ մեկը էկրանին ոչինչ չի տպում, սակայն երկուսն էլ վերագրում են արժեքներ **matric** անունով հիշողության բլոկին.

		0	1	2	3	4	
matric	[0	1	2	3	4	5
		1	2	4	6	8	10
		2	3	6	9	12	15

Մենք օգտագործեցինք նախապրոցեսորի **#define** հրամանը՝ ծրագրի հետագա փոփոխությունները հեշտացնելու նպատակով: Օրինակ՝ եթե 3x5 զանգվածի փոխարեն պահանջվի 4x5 զանգված, ապա ընդհամենը

```
#define HEIGHT 3
```

կդարձնենք

```
#define HEIGHT 4
```

Զանգվածները՝ որպես պարամետրեր

Այժմ քննարկելու ենք, թե ինչպես ֆունկցիային զանգված փոխանցել՝ որպես պարամետր: C++ - ում հնարավոր չէ ֆունկցիային որպես պարամետր փոխանցել հիշողության մի ամբողջ բլոկի արժեքը, նույնիսկ եթե այդ բլոկը իրենից ներկայացնում է զանգված: Դրա փոխարեն ֆունկցիային փոխանցում ենք զանգվածի հասցեն (հիշողության մեջ): Ֆունկցիան, ունենալով զանգվածի հասցեն, ֆունկցիան կարող է կարդալ և փոփոխել այն: Նշենք նաև, որ քանի որ միայն զանգվածի հասցեն է փոխանցվում ֆունկցիային, ֆունկցիայի կանչը կատարվում է շատ արագ՝ անկախ զանգվածի չափերից (զանգվածը չի պատճենվում):

Որպեսզի ֆունկցիան կարողանա ընդունել զանգված, կարող ենք ֆունկցիայի հայտարարության ժամանակ որպես արգումենտ գրել հետևյալը՝

```
տիպ անուն []
```

որտեղ **տիպը** փոխանցվող զանգվածի տիպն է, իսկ **անունը**՝ արգումենտի անունը: Օրինակ՝ հետևյալ ֆունկցիան

```
void funkcia (int arg[])
```

ընդունում է «**int** - ի զանգված» տիպի պարամետր՝ **arg** անունով: Որպեսզի այս ֆունկցիային կարողանանք փոխանցել հետևյալ զանգվածը՝

```
int zangvac [40];
```

բավական է գրել

```
funkcia (zangvac);
```

Օրինակ.

```
// zangvacner, vorpes parametrer
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
    for (int n=0; n<length; n++)
    {
        cout << arg[n] << " ";
    }
    cout << "\n";
}

int main ()
{
    int array1[] = {5, 10, 15};
    int array2[] = {2, 4, 6, 8, 10};
    printarray (array1,3);
    printarray (array2,5);
    return 0;
}
```

```
5 10 15
2 4 6 8 10
```

Ինչպես տեսնում եք, առաջին արգումենտը (**int arg[]**) ընդունում է ցանկացած **int** տիպի զանգված, անկախ դրա չափերից: Այդ պատճառով ավելացրել ենք երկրորդ պարամետրը, որը տեղեկացնելու է ֆունկցիային փոխանցված զանգվածի երկարության մասին: Սա հնարավորություն է տալու մեր **for** ցիկլին իմանալու, թե քանի անգամ այն պետք է կատարվի:

Ֆունկցիային կարելի է փոխանցել նաև բազմաչափ զանգվածներ: Տեսքը եռաչափ զանգվածի համար հետևյալն է՝

տիպ անուն [] [երկարություն1] [երկարություն2]

որտեղ **տիպը** փոխանցվող զանգվածի տիպն է, **անունը**՝ արգումենտի անունը, **երկարություն1** - ը՝ փոխանցվող զանգվածի երկրորդ չափողականության չափը, իսկ **երկարություն2** - ը՝ փոխանցվող զանգվածի երրորդ չափողականության չափը: Օրինակ՝

```
void funkcia (int zangvac[][3][4])
```

Ուշադրություն դարձնենք, որ առաջին փակագծերը դատարկ են, իսկ մնացածը՝ ոչ: Դա կատարվում է այն պատճառով, որ բազմաչափ զանգվածները համակարգչի հիշողության մեջ պահվում են ճիշտ այնպես, ինչպես միաչափները: Թարգմանիչի աշխատանքի ժամանակ լրացուցիչ համարները (ինդեքսները) միավորում է, ինչպես բացատրված է այս բաժնի «Բազմաչափ զանգվածներ» գլխում:

Քիչ փորձ ունեցող ծրագրավորողների մեծամասնությունը հաճախ սխալվում է ֆունկցիային միաչափ կամ բազմաչափ զանգվածներ փոխանցելիս: Այդ պատճառով խորհուրդ է տրվում կարդալ 3.3 բաժինը:

Բաժին 3.2

Սիմվոլային տողեր (Strings of Characters)

Մինչ այժմ հանդիպած բոլոր ծրագրերում մենք օգտագործում էինք միայն թվային փոփոխականներ: Սակայն բացի թվային փոփոխականներից, գոյություն ունեն նաև սիմվոլային տողեր, որոնք թույլ են տալիս ներկայացնել սիմվոլներ, բառեր, նախադասություններ, տեքստեր և այլն: Մինչ այժմ մենք օգտագործում էինք դրանք որպես հաստատուններ, բայց ոչ որպես փոփոխականների արժեքներ:

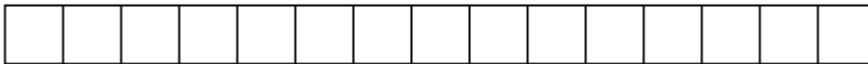
C++ - ում գոյություն չունի փոփոխականի հատուկ տիպ, որը հնարավորություն կտա պահել սիմվոլային տողեր: Այդ պատճառով սիմվոլային տողեր պահելու համար օգտագործում են **char** - երի զանգված (հիշենք, որ **char** տիպն օգտագործվում է մեկ սիմվոլ պահելու համար):

Օրինակ՝ հետևյալ զանգվածը

```
char tox[15];
```

կարող է պահել մինչև 15 սիմվոլ: Պատկերացրեք այն այսպես.

tox



Սակայն պարտադիր չէ, որ բոլոր 15 **char** - երը միշտ սիմվոլ պարունակեն: Օրինակ՝ **tox** - ը սկզբից կարող է պահել “Barev” տողը, իսկ հետո՝ “Inchpes es?” տողը: Որպեսզի սիմվոլային տողում հնարավոր լինի պահել զանգվածի չափից ավելի կարճ տողեր, տողի վերում դնում ենք, այսպես կոչված, **գրոյական սիմվոլով**: Զրոյական սիմվոլի արժեքը **0** է (ավելի հաճախ օգտագործվում է **\0** հաստատունը):

Մենք կարող ենք ներկայացնել **tox** զանգվածը, որը սկզբից պահում է “Barev”, իսկ հետո՝ “Inchpes es?” տողերը հետևյալ կերպ.

tox



Ուշադրություն դարձնենք **գրոյական սիմվոլին** (**\0**): Այն ցույց է տալիս, որ սիմվոլային տողն ավարտվել է: Մուգ ներկված քառակուսիները իրենցից ներկայացնում են անորոշ արժեքներ:

Տողերի սկզբնարժեքավորում

Քանի որ սիմվոլային տողերը զանգվածներ են, նրանց համար գործում են բոլոր այն օրենքները, որոնք գործում են սովորական զանգվածների համար: Օրինակ՝ կարող ենք սիմվոլային տողը սկզբնարժեքավորել հետևյալ կերպ՝

```
char tox[] = { 'B', 'a', 'r', 'e', 'v', '\0' };
```

Այս դեպքում մենք հայտարարեցինք 6 երկարությամբ սիմվոլային տող (**char** – երի 6 - էլեմենտանոց զանգված) և սկզբնարժեքավորեցինք այն “**Barev**” տողը կազմող սիմվոլներով, իսկ վերջում կցեցինք զրոյական սիմվոլը՝ '\0' - ն:

Սակայն գոյություն ունի սիմվոլային տողերի սկզբնարժեքավորման մեկ այլ եղանակ՝ **հաստատուն տողերի** օգտագործումը:

Մեր քննարկած նախորդ օրինակներում արդեն շատ անգամ հանդիպել ենք հաստատուն սիմվոլային տողերի: Դրանք սահմանափակված են " չակերտներով, օրինակ՝

```
"Es grvac em C++ lezvov"
```

արտահայտությունը տողային հաստատուն է:

Ի տարբերություն միավոր չակերտների ('), որոնք օգտագործվում են միավոր սիմվոլային հաստատուններ նշելու համար, կրկնակի չակերտներով (") սահմանափակում ենք տողային հաստատուններ: Այն տողերին, որոնք գրված են կրկնակի չակերտների մեջ, ավտոմատ կցվում է զրոյական սիմվոլը՝ '\0':

Ստացվեց, որ մենք կարող ենք սկզբնարժեքավորել **tox** տողային հաստատունը հետևյալ երկու եղանակներով՝

```
char tox[] = { 'B', 'a', 'r', 'e', 'v', '\0' };  
char tox[] = "Barev";
```

Այս երկու տողերն իրար համարժեք են:

Նշենք նաև այն փաստը, որ տողային փոփոխականին արժեք վերագրումը վերը նշված եղանակներով ճիշտ է միայն այդ փոփոխականի սկզբնարժեքավորման ժամանակ: Ասվածը նշանակում է, որ ճիշտ չեն ստորը բերված արտահայտությունները.

```
tox = "Barev";  
tox[] = "Barev";  
tox = { 'B', 'a', 'r', 'e', 'v', '\0' };
```

Այսպիսով հիշենք. սիմվոլային զանգվածին կարող ենք վերագրել տողային հաստատուն սիմվոլային զանգվածի սկզբնարժեքավորման ժամանակ:

Արժեքների վերագրումը տողերին

Քանի որ վերագրման **ձախ արժեքը** (lvalue) կարող է լինել միայն զանգվածի տարր, այլ ոչ թե զանգվածն ամբողջությամբ, ապա ճիշտ է օգտագործել հետևյալ մեթոդը՝ **char** - երի զանգվածին սիմվոլային տող վերագրելու համար.

```
tox[0] = 'B';  
tox[1] = 'a';  
tox[2] = 'r';
```

```
tox[3] = 'e';
tox[4] = 'v';
tox[5] = '\\0';
```

Բայց ինչպես տեսնում եք, սա գործնական մեթոդ չէ: Զանգվածներին և մասնավորապես, սիմվոլային տողերին արժեքներ վերագրելու համար կան մի շարք ֆունկցիաներ, ինչպես, օրինակ, **strcpy** - ն: **strcpy** (**string copy**, թարգմանաբար՝ տողի պատճենում) ֆունկցիան հայտարարված է **cstring** (անցյալում՝ string.h) գրադարանում և կարող է կանչվել հետևյալ կերպ.

```
strcpy (տող1, տող2);
```

Այն պատճենում է **տող2** - ի պատրունակությունը **տող1** - ի մեջ: Այստեղ **տող2** - ը կարող է լինել զանգված, ցուցիչ կամ տողային հաստատուն: Հետևյալ արտահայտությունը **tox** սիմվոլային տողին կվերագրի **"Barev"** տողային հաստատունը.

```
strcpy (tox, "Barev");
```

Օրինակ.

```
// arjeqi veragrum@ toxin
#include <iostream>
#include <cstring>
using namespace std;

int main ()
{
    char harc [20];
    strcpy (harc, "Inchpes es?");
    cout << harc;
    return 0;
}
```

Inchpes es?

Զմոռանանք ներառել **<cstring>** գրադարանը **strcpy** ֆունկցիան օգտագործելիս:

Սակայն մենք կարող ենք գրել **strcpy** ֆունկցիայի նման ֆունկցիա ինքներս.

```
// arjeqi veragrum@ toxin
#include <iostream>
using namespace std;

void setstring (char szOut [], char szIn [])
{
    int n=0;
    do {
        szOut[n] = szIn[n];
    } while (szIn[n++] != '\\0');
}

int main ()
{
```

```
char harc [20];
setstring (harc,"Inchpes es?");
cout << harc;
return 0;
}
```

Inchpes es?

Չանգվածին արժեքներ վերագրելու մյուս տարածված ձևն է մուտքի հոսքի օգտագործումը (**cin**): Այս դեպքում տողային հաստատունի արժեքը վերագրվում է օգտագործողի կողմից՝ ծրագրի աշխատանքի ընթացքում:

cin մուտքի հոսքի միջոցով սիմվոլային տողին արժեք տալիս սովորաբար օգտագործվում է մուտքի հոսքի **getline** մեթոդը, որը կարող է կանչվել՝ համաձայն հետևյալ նկարագրության.

```
cin.getline(char բուֆեր[], int երկարություն, char տարանջատիչ = '\n');
```

որտեղ **բուֆերը** հասցե է, որը ցույց է տալիս, թե որտեղ պետք է պահվի մուտքագրված արժեքը (օրինակ՝ զանգված), **երկարությունը՝ բուֆերի** առավելագույն երկարությունն է (զանգվածի չափը), իսկ **տարանջատիչը** սիմվոլ է, որն օգտագործվում է մուտքի ավարտը որոշելու համար. լռությամբ այդ սիմվոլը՝ նոր տողի **'\n'** սիմվոլն է:

Հետևյալ օրինակը կրկնում է այն ամենը, ինչ հավաքվում է ստեղծաբանի վրա: Սա շատ պարզ ծրագիր է, սակայն ծառայում է լավ օրինակ, թե ինչպես օգտագործել **cin.getline** - ը տողերի հետ.

```
// cin-@ toxeri het
#include <iostream>
using namespace std;

int main ()
{
    char buffer [100];
    cout << "Inchpes e dzer anun@? ";
    cin.getline (buffer,100);
    cout << "Barev " << buffer << ".\n";
    cout << "Vorn e dzer sirac tim@? ";
    cin.getline (buffer,100);
    cout << "Es sirum em " << buffer << "-@ evs.\n";
    return 0;
}
```

Inchpes e dzer anun@? Suren
Barev Suren.
Vorn e dzer sirac tim@? Manchester United
Es sirum em Manchester United-@ evs.

Ուշադրություն դարձնենք այն բանին, որ **cin.getline** - ի երկու կանչերի ժամանակ էլ մենք օգտագործել ենք միևնույն զանգվածը (**buffer**): **cin.getline** ֆունկցիայի երկրորդ կանչի ժամանակ **buffer** - ի պարունակությունը չի ջնջվում, սակայն նոր սիմվոլային տողը գրվում է՝ **buffer** - ի առաջին (զրո համարով) տարրից սկսած:

Բայց քանի որ սիմվոլային տողն ավարտվում է զրոյական սիմվոլով, ապա, այդ սիմվոլից սկսած, բոլոր մնացածները անտեսվում են, նույնիստ եթե դրանք ներկա են:

Եթե դուք հիշում եք, «Հաղորդակցություն օգտագործողի հետ» բաժնում ստանդարտ հոսքից ինֆորմացիա կարդալու համար օգտագործում էինք >> օպերատորը: Այս մեթոդը ևս կարող է օգտագործվել **cin.getline** – ի փոխարեն՝ սիմվոլային տողեր կարդալու համար: Օրինակ՝ մեր ծրագրի այն մասերը, որտեղ պահանջում էինք օգտագործողի կողմից արժեքի մուտքագրում, կարող ենք փոխարինել հետևյալ արտահայտությամբ.

```
cin >> buffer;
```

Սակայն այդ մեթոդը ունի որոշ սահմանափակումներ, որոնք չեն գործում **cin.getline** – ի դեպքում:

- Այն կարող է օգտագործվել միայն բառեր (ոչ ամբողջ նախադասություններ) կարդալու համար: Պատճառն այն է, որ նրա համար, որպես **տարանջատիչ** ծառայում են դատարկ սիմվոլները՝ բացատանիշերը (space), տարուլացիաները և նոր տողերը:
- Չի թույլատրվում նշել **բուֆերի** երկարությունը: Սա դարձնում է ծրագիրը ոչ կայուն այն դեպքում, երբ օգտագործողը մուտքագրում է ընդունող զանգվածի չափերից (մեր դեպքում՝ **100**) ավելի երկար արտահայտություն:

Այդ պատճառով, **cin** – ից սիմվոլային տողեր պահանջելիս, խորհուրդ է տրվում **cin >>** - ի փոխարեն օգտագործել **cin.getline** - ը:

Տողերի փոխակերպումն այլ տիպերի

Քանի որ տողում կարող են պահվել ցանկացած տիպի տվյալներ, օրինակ՝ թվեր, ապա հարմար կլինի կարողանալ վերածել տողի պարունակությունը թվային հաստատունի: Օրինակ՝ տողը կարող է պահել "1977", որն իրենից ներկայացնում է 5 սիմվոլից կազմված զանգված: Սակայն այն այնքան էլ հեշտ չէ վերածել թվային տիպի: Այդ նպատակով **cstdlib** (**stdlib.h**) գրադարանը ապահովում է մեզ հետևյալ երեք ֆունկցիաներով.

- **atoi**՝ վերածում է տողը **int** տիպի.
- **atol**՝ վերածում է տողը **long** տիպի.
- **atof**՝ վերածում է տողը **float** տիպի:

Բոլոր այս ֆունկցիաներն ընդունում են մեկ պարամետր և վերադարձնում են արժեք՝ ներկայացված տվյալ տիպով (**int**, **long** կամ **float**): Այս ֆունկցիաները, **cin** - ի **getline** մեթոդի հետ, ավելի արդյունավետ են օգտագործողի կողմից մուտքագրված թիվը կարդալու համար, քան "cin >>" մեթոդը.

```
// cin ev ato* funkcianer@
#include <iostream>
#include <stdlib.h>
using namespace std;

int main ()
{
    char buffer [100];
    float gin;
    int qanak;
    cout << "Mutqagrek gin@: ";
    cin.getline (buffer,100);
    gin = atof (buffer);
    cout << "Mutqagrek qanak@: ";
    cin.getline (buffer,100);
    qanak = atoi (buffer);
    cout << "@ndhanur gumar@: " << gin*qanak;
    return 0;
}
```

```
Mutqagrek gin@: 2.75
Mutqagrek qanak@: 21
@ndhanur gumar@: 57.75
```

Բաժին 3.3

Ցուցիչներ (Pointers)

Մենք արդեն գիտենք, որ փոփոխականները պահվում են համակարգչի հիշողության բջիջներում, որոնց կարող ենք դիմել նույնարկիչներով: Համակարգչի հիշողության բջիջներից յուրաքանչյուրն ունի կոնկրետ հասցե, որը համապատասխանում է միայն այդ բջիջին:

Համակարգչի հիշողության հասցեները սովորական թվեր են:

Հասցեավորման (Address) օպերատոր (&)

Երբ մենք հայտարարում ենք որևէ փոփոխական, այն պետք է պահվի համակարգչի հիշողության մեջ՝ որևէ կոնկրետ հասցեում: Բարեբախտաբար սա արվում է ավտոմատաբար՝ մենք չենք որոշում, թե որտեղ պետք է պահվի նոր փոփոխականը: Դա անում է օպերացիոն համակարգը, սակայն երբեմն մեզ կարող է հետաքրքրել, թե ինչ հասցեում է պահված մեր փոփոխականը:

Փոփոխականի հասցեն կարելի է ստանալ՝ փոփոխականի նույնարկիչից առաջ դնելով հասցեավորման **&** նշանը, որը նշանակում է «դրա հասցեն»: Օրինակ՝

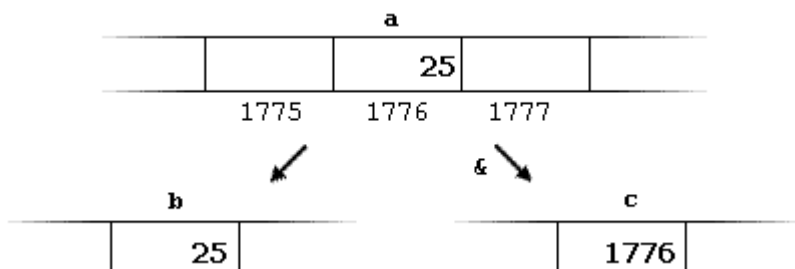
```
a = &b;
```

Այս տողով **a** փոփոխականի մեջ կգրի **b** - ի հասցեն: **b** - ից առաջ դնելով **&** նշանը՝ մենք ասում ենք, որ ի նկատի ունենք փոփոխականի հասցեն, այլ ոչ թե նրա արժեքը:

Ենթադրենք, որ **a** փոփոխականը պահվել է **1776** հասցեում, և մենք գրում ենք հետևյալը՝

```
a = 25;  
b = a;  
c = &a;
```

Արդյունքը ցույց է տրված հետևյալ գծագրում՝



Այստեղ **b** փոփոխականին տվեցինք **a** փոփոխականի արժեքը, ինչպես շատ անգամ արել ենք նախորդ օրինակներում: Իսկ **c** փոփոխականին տվեցինք հիշողության այն հասցեն, որտեղ պահված է **a** փոփոխականը (այդ հասցեն, ըստ մեր ենթադրության, **1776** է):

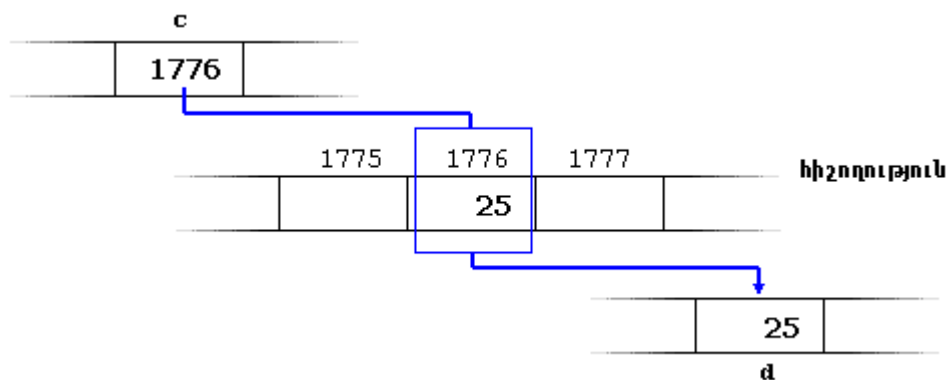
Փոփոխականին, որը պահում է մեկ այլ փոփոխականի հասցե (կոնկրետ օրինակում **c** - ն պահում էր **a** - ի հասցեն), մենք կանվանենք ցուցիչ (**pointer**): Ցուցիչները C++ լեզվի կարևորագույն հասկացություններից են և ունեն շատ մեծ կիրառություն: Շուտով կտեսնենք, թե ինչպես հայտարարել այդ տիպի փոփոխականներ:

Հետհասցեավորման (Reference) օպերատոր (*)

Օգտագործելով ցուցիչներ՝ մենք կարող ենք ուղղակիորեն դիմել այդ ցուցիչի ցույց տված հասցեում պահվող արժեքին: Դրա համար ցուցիչից առաջ դնում ենք հետհասցեավորման նշան (*), որը նշանակում է «արժեքը, որը ցույց է տալիս սրված ցուցիչը»: Եթե, ունենալով նախորդ օրինակի փոփոխականները, գրենք՝

```
d = *c;
```

d - ն կստանա **25** արժեքը, քանի որ **c** - ն **1776**, իսկ **1776** հասցեում պահված է **25** թիվը:



Մեկ անգամ էլ նշենք, որ **c** - ի արժեքը **1776** է, իսկ ***c** - ն համապատասխանում է **1776** հասցեում պահված արժեքին՝ **25** - ին:

```
d = c; // d-n havasar e c-in (1776)
d = *c; // d-n havasar e c-i cuyc tva& arjeqin (25)
```

Այժմ պետք է լիովին պարզ լինի, որ եթե նախորդ օրինակում գրված է

```
a = 25;
c = &a;
```

ապա ճիշտ են հետևյալ արտահայտությունները՝

```
a == 25
&a == 1776
c == 1776
*c == 25
*c == a
```

«Ցուցիչ» տիպի փոփոխականների հայտարարումը

Քանի որ ցուցիչի միջոցով կարող ենք ուղղակիորեն դիմել նրա ցույց տված արժեքին, ապա ցուցիչը հայտարարելիս անհրաժեշտ է դառնում նշել, թե ինչ տիպի փոփոխականի է ցույց տալու ցուցիչը: **char** տիպի փոփոխականին ցույց տալը նույնը չէ, ինչ **int** կամ **float** ցույց տալը:

Այսպիսով՝ ցուցիչի հայտարարման գրելաձևը հետևյալն է՝

```
տիպ * ցուցիչ_անուն;
```

որտեղ **տիպ** այն տվյալի տիպն է, որի վրա ցույց է տալու ցուցիչը (սա չի կարելի խառնել ցուցիչի տիպի հետ): Օրինակ՝

```
int * tiv;  
char * tar;  
float * mectiv;
```

Սրանք երեք ցուցիչների հայտարարություններ են, որոնք ցույց են տալիս տարբեր տիպերի վրա: Բայց դրանք իրականում զբաղեցնում են հավասար քանակությամբ հիշողություն՝ չնայած, որ տարբեր տիպերի են:

Աստղանիշը (*), որը գրվում է ցուցիչը հայտարարելիս, պարզապես նշում է, որ հայտարարվողը ցուցիչ է. այն ոչ մի ընդհանուր բան չունի հետհասցեավորման օպերատորի հետ, որը նույնպես գրվում է աստղանիշով:

```
// im arachin cucich@  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    int arjeq1 = 5, arjeq2 = 15;  
    int * im_cucich;  
  
    im_cucich = &arjeq1;  
    *im_cucich = 10;  
    im_cucich = &arjeq2;  
    *im_cucich = 20;  
    cout << "arjeq1==" << arjeq1 << " / arjeq2==" << arjeq;  
    return 0;  
}
```

```
arjeq1==10 / arjeq2==20
```

Ուշադրություն դարձնենք, թե ինչպես **arjeq1** և **arjeq2** փոփոխականների արժեքները անուղղակիորեն փոխվեցին: Սկզբում մենք **im_cucich** - ին տվեցինք **arjeq1** - ի հասցեն՝ օգտագործելով հասցեավորման օպերատորը (&): Այնուհետև **im_cucich** - ի ցույց տված արժեքը դարձրեցինք **10**. **im_cucich** - ը ցույց էր տալիս **arjeq1** - ին, և հետևաբար մենք փոխեցինք **arjeq1** - ը:

Որպեսզի ցուցադրենք, որ ծրագրի մեջ կարելի է փոխել ցուցիչի ցույց տված հասցեն, նույն գործողությունը կատարեցինք **arjeq2** - ի և միևնույն ցուցիչի հետ:

Ահա մի քիչ ավելի բարդ օրինակ.

```
// eli cucichner
#include <iostream>
using namespace std;

int main ()
{
    int arjeq1 = 5, arjeq2 = 15;
    int *p1, *p2;

    p1 = &arjeq1; // p1 = arjeq1-i hascein
    p2 = &arjeq2; // p2 = arjeq2-i hascein
    *p1 = 10;      // p1-i cuyc tva& arjeq@ = 10
    *p2 = *p1;    // p2-i cuyc tva& arjeq@ = p1-i cuyc tva& arjeq@in

    p1 = p2;      // p1 = p2 (cucichneri arjeqner@ artagrvm en)
    *p1 = 20;     // p1-i cuyc tva& arjeq@ = 20

    cout << "arjeq1==" << arjeq1 << " / arjeq2==" << arjeq2;
    return 0;
}

arjeq1==10 / arjeq2==20
```

const –ի օգտագործումը ցուցիչներում

Այս բանալի բառը մենք կարող ենք օգտագործել հաստատուն (const) փոփոխականի վրա և հաստատուն հասցեի վրա ցույց տվող ցուցիչներ հայտարարելու համար:

Հաստատուն փոփոխականի վրա ցուցիչ հայտարարելու գրելաձևն սա է.

```
const տիպ * ցուցիչ_անուն;
```

Դիտարկենք հետևյալ օրինակը.

```
const int x = 7;
int *y = &x;
```

Մրա վրա թարգմանիչը կտա սխալ: Պատճառն այն է, որ թարգմանիչը այս տողը թարգմանելիս, չի ստուգում, թե արդյոք, հետագայում, *y* ցուցիչի միջոցով, փոփոխվում է *x* հաստատունի արժեքը, թե՞ ոչ: Այդ պատճառով ցուցիչ հայտարարելիս գրում են `const` բանալի բառը՝ ցուցիչի ցույց տվող տիպից առաջ.

```
const int x = 7;
const int *y = &x;
```

Սակայն այս դեպքում `const`-ը ոչ մի սահմանափակում չի դնում ցուցիչի հասցեի փոփոխման վրա, այսինքն մենք կարող ենք գրել՝

```
int z = 5;
y = &z;
```

y-ը այժմ ցույց կտա z-ի վրա, սակայն, չնայած, որ z-ը հաստատուն չէ, մենք չենք կարող փոփոխել z-ի արժեքը y-ի միջոցով, քանի որ ցուցիչը հայտարարված էր հաստատուն տիպի վրա

```
*y = 7; //Sxal! y cucich@ const e!
```

const-ը կարող է օգտագործվել նաև հաստատուն հասցեի վրա ցուցիչներ հայտարարելու համար: Դրա գրելաձևն սա է.

```
տիպ * const ցուցիչ_անուն;
```

Օրինակ, որպեսզի չթույլատրենք մեր նախորդ օրինակում y ցուցիչի փոփոխումը, պետք է այն հայտարարենք այսպես.

```
const int* const y = &x;
```

Որից հետո էլ արդեն հետևյալ արտահայտությունը կդառնա սխալ՝

```
y = &z;
```

Այսպիսով const բանալի բառը ունի երկու կիրառություն ցուցիչների դեպքում և դրա բնույթը կախված է ցուցիչի հայտարարման մեջ դրա տեղից:

Ցուցիչներ և զանգվածներ

Չանգվածների հասկացությունը խիստ կապված է ցուցիչների հասկացության հետ: Չանգվածի նույնարկիչը իրականում ցուցիչ է զանգվածի առաջին տարրին: Այսպիսով՝ ցուցիչներն ու զանգվածները համարժեք հասկացություններ են:

```
int tver [20];
int * p;
```

Հետևյալ տողը թույլատրելի հրաման է՝

```
p = tver;
```

Այս տողից սկսած՝ **p** - ն և **tver** - ը համարժեք են և ունեն նույն հատկությունները: Միակ տարբերությունն այն է, որ **p** - ին կարող ենք տալ նոր արժեք, քանի որ այն ցուցիչ է, իսկ **tver** - ը միշտ ցույց կտա **20** թվերից առաջինը: Այսպիսով, ի տարբերություն **p** - ի, որը փոփոխական ցուցիչ է, **tver** - ը հաստատուն ցուցիչ է: Չնայած նախորդ արտահայտությունը թույլատրելի էր՝ հետևյալը թույլատրելի չէ.

```
tver = p;
```

Ահա ցուցիչների վերաբերյալ ևս մի օրինակ.

```
// eli cucichner
#include <iostream>
using namespace std;

int main ()
{
    int tver[5];
    int * p;
    p = tver; *p = 10;
    p++; *p = 20;
    p = &tver[2]; *p = 30;
    p = tver + 3; *p = 40;
    p = tver; *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << tver[n] << ", ";
    return 0;
}
```

10, 20, 30, 40, 50,

«Չանգվածներ» թեմայում մենք օգտագործում էինք փակագծեր՝ [], որպեսզի նշենք զանգվածի այն տարրի համարը, որին ուզում ենք դիմել: Բայց դա նույնն է, ինչ ցուցիչի հասցեին գումարել անհրաժեշտ թիվը: Օրինակ՝ հետևյալ երկու արտահայտությունները՝

```
a[5] = 0; // a [5-rd andam@] = 0
*(a+5) = 0; // (a+5)-i cuyc tva&& = 0
```

բոլորովին համարժեք են և՛ ցուցիչ, և՛ զանգված **a** - ի դեպքում:

Ցուցիչների սկզբնարժեքավորում

Ցուցիչ հայտարարելիս կարելի է այն միանգամից սկզբնարժեքավորել՝ նշել հասցեն, որն ուզում ենք այն ցույց տա:

```
int tiv;
int *cucich = &tiv;
```

սա համարժեք է հետևյալին.

```
int tiv;
int *cucich;
cucich = &tiv;
```

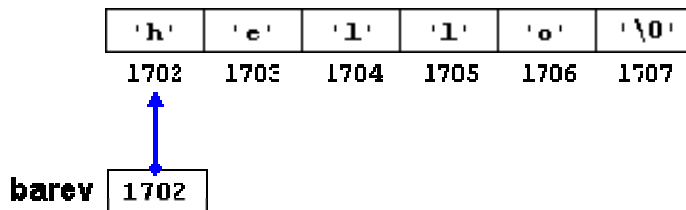
Երբ ցուցիչին վերագրում ենք որևէ արժեք, այդ արժեքը վերագրվում է ոչ թե նրանում գրված հասցեով հիշողության տիրույթին, այլ որոշում է թե ինչ հասցե է ցույց տալիս այդ ցուցիչը: Այսպիսով պետք է զգույշ լինել, որ չխառնել նախորդ գրվածը հետևյալի հետ՝

```
int tiv;
int *cucich;
*cucich = &tiv;
```

Ինչպես և զանգվածների դեպքում, թարգմանիչը հնարավորություն է տալիս ցուցիչները սկզբնարժեքավորել հաստատուններով.

```
char * barev = "hello";
```

Այս դեպքում հիշողության մեջ տեղ է զբաղեցվում **"hello"** հաստատունը պահելու համար և հիշողության այդ հատվածի առաջին տարրի հասցեն տրվում է **barev** ցուցիչին: Եթե ենթադրենք, որ **"hello"** տողը պահվում է **1702** և նրան հաջորդող հասցեներում, ապա նախորդ հայտարարումը կունենա հետևյալ տեսքը`



Կարևոր է նշել, որ **barev** - ը պարունակում է **1702** արժեքը, այլ ոչ թե **'h'** և ոչ էլ **"hello"**, իսկ **1702** - ը այս սիմվոլներից առաջինի՝ **'h'** - ի հասցեն է:

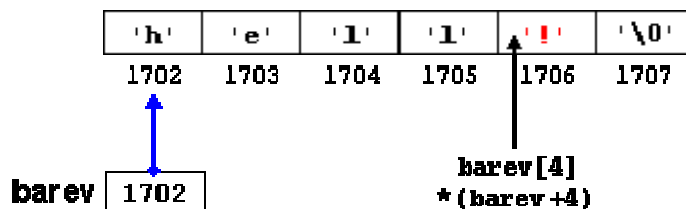
barev փոփոխականը ցույց է տալիս սիմվոլների տողը և կարող է օգտագործվել ճիշտ այնպես, ինչպես ցանկացած զանգված: Օրինակ՝ եթե մեր տրամադրությունը փոխվի, և որոշենք **'o'** - ն փոխարինել **'!'** - ով (hell նշանակում է դժողք), կարող ենք վարվել հետևյալ կերպ.

```
barev[4] = '!';
```

կամ

```
*(barev+4) = '!';
```

Երկու դեպքում էլ տեղի կունենա հետևյալը`



Ցուցիչների թվաբանությունը

Ցուցիչների թվաբանությունը տարբերվում է մյուս ամբողջ տիպերի (**int**, **long**, ...) հետ կատարվող թվաբանությունից: Սկսենք նրանից, որ ցուցիչների համար սահմանված են միայն գումարման և հանման գործողությունները: Մնացած գործողությունները պարզապես իմաստ չունեն ցուցիչների աշխարհում: Իսկ

սահմանված գումարման և հանման գործողությունների աշխատանքը կախված է ցուցիչի տիպից:

Հիշենք, որ տվյալների տարբեր տիպեր տարբեր քանակությամբ հիշողություն են զբաղեցնում. **char** - ը զբաղեցնում է 1 բայթ, **short** - ը՝ 2 բայթ, իսկ **long** - ը՝ 4 բայթ: Դիցուք ունենք 3 ցուցիչ՝

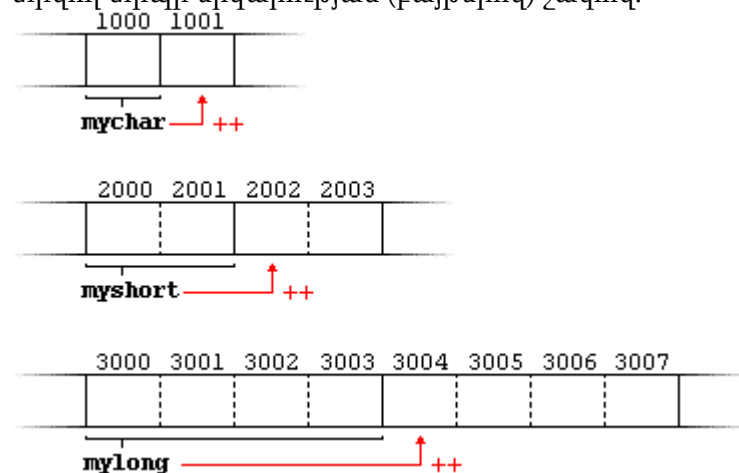
```
char *mychar;  
short *myshort;  
long *mylong;
```

և մենք գիտենք, որ դրանք համապատասխանաբար ցույց են տալիս հիշողության 1000, 2000 և 3000 հասցեները:

Այսպիսով, եթե մենք գրենք

```
mychar++;  
myshort++;  
mylong++;
```

mychar - ը, ինչպես կարելի էր ենթադրել, կպարունակի 1001 արժեքը, սակայն **myshort** - ը կպարունակի 2002 արժեքը, իսկ **mylong** - ը՝ 3004 արժեքը: Պատճառն այն է, որ երբ ցուցիչը մեծացնում ենք 1 - ով, նրան ստիպում ենք ցույց տալ նույն տիպի հաջորդ տարրը: Հետևաբար ցուցիչի արժեքը մեծանում է ցույց տրվող տիպի երկարության (բայթերով) չափով:



Սա ճիշտ է և՛ ցուցիչները մեծացնելիս, և՛ փոքրացնելիս: Տեղի կունենար ճիշտ նոյնը, եթե գրեինք

```
mychar = mychar + 1;  
myshort = myshort + 1;  
mylong = mylong + 1;
```

Պետք է հիշել, որ և՛ աճման (++) , և՛ նվազման (--) օպերատորներն ավելի բարձր նախապատվությամբ գործողություններ են, քան հետհասցեավորման (*) օպերատորը: Հետևյալ արտահայտությունները կարող են անհասկանալի լինել.

```
*p++;  
*p++ = *q++;
```

Առաջինը համարժեք է `*(p++)` - ին. սրանում մեծանում է `p` - ի ցույց տված հասցեն, այլ ոչ թե այնտեղ գրված արժեքը: Մյուսը համարժեք է հետևյալին.

```
*p = *q;  
p++;  
q++;
```

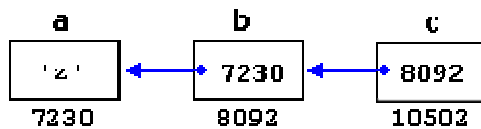
Ինչպես միշտ՝ խորհուրդ ենք տալիս օգտագործել փակագծեր՝ `()`:

Ցուցիչ՝ ցուցիչի վրա

`C++` - ը հնարավորություն է տալիս օգտագործել ցուցիչներ, որոնք ցույց են տալիս այլ ցուցիչների, որոնք էլ, իրենց հերթին, ցույց են տալիս այլ տվյալների: Դա անելու համար պարզապես անհրաժեշտ է ավելացնել աստղանիշ `*`՝ հետհասցեավորման ամեն մակարդակի համար.

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

Եթե համարենք, որ `a`, `b`, `c` փոփոխականները պահվել են համապատասխանաբար 7230, 8092 և 10502 հասցեներում, ապա վերևում գրվածը կարելի է նկարագրել այսպես.



(վանդակների մեջ գրված են փոփոխականների արժեքները, վանդակների տակ՝ նրանց հասցեները):

void ցուցիչներ

Ցուցիչի **void** տիպը յուրահատուկ տիպ է: **void** տիպի ցուցիչները կարող են ցույց տալ ցանկացած տիպի փոփոխականների՝ ամբողջ թվերի, սիմվոլային տողերի և այլն: Միակ սահմանափակումը այն է, որ ցույց տրվող արժեքին չենք կարող անմիջականորեն դիմել (չենք կարող օգտագործել հետհասցեավորման `*` օպերատորը): Այս պատճառով միշտ ստիպված կլինենք կատարել տիպի ձևափոխում (type casting), որպեսզի **void** ցուցիչը դարձնենք կոնկրետ տիպի ցուցիչ, որի հետ կարող ենք հանգիստ աշխատել:

Քննարկենք մի օրինակ.


```

// amboxch tiv achacnox
#include <iostream>
using namespace std;

void mecacnel (void* data, int type)
{
    switch (type)
    {
        case sizeof(char) : (*((char*)data)++)++; break;
        case sizeof(short): (*((short*)data)++)++; break;
        case sizeof(long) : (*((long*)data)++)++; break;
    }
}

int main ()
{
    char a = 5;
    short b = 9;
    long c = 12;
    mecacnel (&a,sizeof(a));
    mecacnel (&b,sizeof(b));
    mecacnel (&c,sizeof(c));
    cout << (int) a << ", " << b << ", " << c;
    return 0;
}

```

6, 10, 13

sizeof - ը C++ - ում պարունակվող օպերատորներից է: Այն վերադարձնում է իր արգումենտի չափը՝ բայթերով: Օրինակ՝ **sizeof(char)** - ը 1 է, քանի որ **char** տիպի երկարությունը 1 բայթ է:

Տուցիչ ֆունկցիային

C++ - ը հնարավորություն է տալիս աշխատել ֆունկցիայի ցույց տվող ցուցիչների հետ: Սրա կարևորագույն գործածություններից մեկը ֆունկցիային մեկ այլ ֆունկցիա որպես արգումենտ փոխանցելն է: Տունկցիային ցույց տվող ցուցիչը հայտարարվում է ֆունկցիայի նախատիպի պես, այն բացառությամբ, որ ֆունկցիայի անունը գրում ենք փակագծերի մեջ և նրանից առաջ դնում աստղանիշ:

```

// cucich funkciayi vra
#include <iostream>
using namespace std;

int gumarum (int a, int b)
{ return (a+b); }

int hanum (int a, int b)
{ return (a-b); }

int (*minus)(int,int) = hanum;

```

```

int gorcoxutyun (int x, int y, int (*inchkanchem)(int,int))
{
    int g;
    g = (*inchkanchem)(x,y);
    return (g);
}

int main ()
{
    int m,n;
    m = gorcoxutyun (7, 5, gumarum);
    n = gorcoxutyun (20, m, minus);
    cout <<n;
    return 0;
}

```

8

Այս օրինակում **minus** - ը գլոբալ ցուցիչ է, որը ցույց է տալիս երկու **int** տիպի արգումենտ ընդունող ֆունկցիայի: Այն սկզբնարժեքավորել ենք **hanum** ֆունկցիայով:

```
int (* minus)(int,int) = hanum;
```

Բաժին 3.4

Դինամիկ հիշողություն (Dynamic Memory)

Մինչ այժմ մեր բոլոր ծրագրերում վերցնում էինք այնքան հիշողություն, որքան անհրաժեշտ էր մեր հայտարարած զանգվածների, փոփոխականների և այլ օբյեկտների համար: Եվ բոլոր դեպքերում հիշողության չափը որոշված էր դեռևս ծրագրի աշխատանքից առաջ: Այս տիպի հիշողությունը կոչվում է **ստատիկ** հիշողություն: Բայց շատ հաճախ կրող ենք նախապես չիմանալ, թե ծրագրին աշխատանքի համար ինչքան հիշողություն անհրաժեշտ կլինի: Անհրաժեշտ հիշողության չափը կարող է կախված լինել օգտագործողի մուտքագրված տվյալներից: Օրինակ՝ հնարավոր է, որ հարկավոր լինի ծրագրի աշխատանքի ընթացքում որոշել որևէ զանգվածի չափ:

Նման խնդիրների համար օգտագործվում է **դինամիկ հիշողություն**: Դինամիկ հիշողության հետ աշխատանքը կատարվում է **C++** - ում առկա **new** և **delete** օպերատորների միջոցով:

new և **new []** օպերատորները

Դինամիկ հիշողություն պահանջելու համար օգտագործում են **new** օպերատորը: **new** - ն հաջորդվում է փոփոխականի տիպով և, անհրաժեշտության դեպքում, քառակուսի փակագծերի մեջ գրված տարրերի քանակով: Նրա տեսքը հետևյալն է՝

```
ցուցիչ = new տիպ
```

կամ

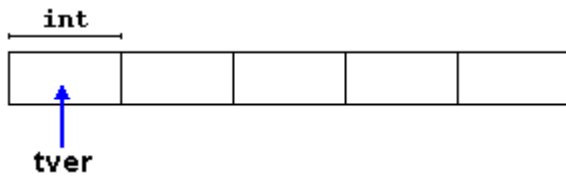
```
ցուցիչ = new տիպ [ տարրերի քանակ ]
```

Առաջին արտահայտությունն օգտագործվում է այն ժամանակ, երբ հարկավոր է լինում հատկացնել մեկ տարրից կազմված **տիպ** տիպի դինամիկ հիշողություն, իսկ երկրորդը՝ **տիպ** տիպի տարրերի բլոկ (զանգված) հատկացնելու համար:

Օրինակ՝

```
int * tver;  
tver = new int [5];
```

Այս դեպքում օպերացիոն համակարգը **int** տիպի 5 տարրերի համար հատկացնում է հիշողություն և **tver** ցուցիչին վերադարձնում է հատկացված հիշողության սկզբնական հասցեն (հատկացված զանգվածի առաջին տարրի հասցեն հիշողության մեջ): Հիշեք նաև, որ զանգվածի տարրերը հիշողության մեջ դասավորված են հաջորդաբար.



Կարող է հարց առաջանալ, թե ինչու՞մն է տարբերությունը սովորական զանգվածի հայտարարության և ցուցիչին՝ **new** - ով հիշողության հասցե վերագրելու միջև: Ամենակարևոր տարբերությունն այն է, որ սովորական զանգվածի չափը պետք է լինի հաստատուն արժեք, որը սահմանափակում է մեր ծրագրի հնարավորությունները, մինչդեռ դինամիկ հիշողության անջատման ժամանակ կարող ենք պահանջել այնքան հիշողություն, որքան անհրաժեշտ է, և հիշողության չափը արտահայտել փոփոխականներով, հաստատուններով կամ էլ դրանցից կազմված այլ արտահայտություններով:

Դինամիկ հիշողությունը կառավարվում է օպերացիոն համակարգի կողմից: Ցանկացած ծրագիր, որին անհրաժեշտ է դինամիկ հիշողություն, դիմում է օպերացիոն համակարգին: Օպերացիոն համակարգը անջատում է անհրաժեշտ քանակությամբ հիշողություն և համապատասխան հասցեն փոխանցում ծրագրին: Սակայն կարող է առաջանալ մի դեպք, երբ օպերացիոն համակարգը չունենա ազատ հիշողություն: Այդ դեպքում **new** օպերատորը կվերադարձնի **զրոյական (null)** ցուցիչ: Այդ պատճառով միշտ խորհուրդ է տրվում դինամիկ հիշողություն հատկացնելիս ստուգել ցուցիչի արժեքը.

```
int * tver;
tver = new int [5];
if (tver == NULL) {
    // hishoxutiun chka!
}
```

delete օպերատորը

Քանի որ համակարգչի հիշողությունը անվերջ չէ, ապա երբ այն մեր ծրագրին այլևս հարկավոր չէ, այն պետք է ազատվի դինամիկ հիշողության հետագա կանչերի նպատակով: **delete** օպերատորը ստեղծված է այդ նպատակով: Դրա տեսքն է.

```
delete ցուցիչ;
```

կամ

```
delete [] ցուցիչ;
```

Առաջին արտահայտությունը պետք է օգտագործվի միավոր տարրին, իսկ երկրորդը՝ մի քանի տարրերին (զանգվածին) հատկացված հիշողությունը ազատելու համար:

```

#include <iostream>
using namespace std;
#include <stdlib.h>

int main ()
{
    char input [100];
    int i, n;
    long * l;
    cout << "Qani hat tiv eq duq uzum mutqagrel? ";
    cin.getline (input, 100);
    i = atoi (input);
    l = new long[i];
    if (l == NULL) exit (1);
    for (n=0; n<i; n++)
    {
        cout << "Mutqagrek tiv: ";
        cin.getline (input,100); l[n]=atol (input);
    }
    cout << "Duc mutqagrecik: ";
    for (n=0; n<i; n++)
        cout << l[n] << ", ";
    delete[] l;
    return 0;
}

```

```

Qani hat tiv eq duq uzum mutqagrel? 5
Mutqagrek tiv: 75
Mutqagrek tiv: 436
Mutqagrek tiv: 1067
Mutqagrek tiv: 8
Mutqagrek tiv: 32
Duc mutqagrecik: 75, 436, 1067, 8, 32,

```

Այս պարզ օրինակը, որը հիշում է թվեր, չունի որևէ սահմանափակում թվերի քանակի վրա, այսինքն՝ կարող է հիշել այնքան թիվ, որքան օգտագործողը ցանկանա (իհարկե եթե գոյություն ունենա այդքան ազատ հիշողություն):

NULL հաստատունի արժեքը որոշված է **C++** - ի շատ գրադարաններում՝ գրոյական ցուցիչը ներկայացնելու նպատակով: Այն դեպքում, եթե այն որոշված չլինի, կարող ենք որոշել այն հետևյալ արտահայտությամբ.

```
#define NULL 0
```

Զրոյական ցուցիչներ ստուգելիս տարբերություն չկա **0** կամ **NULL** օգտագործելու միջև, սակայն ցուցիչների համար խորհուրդ է տրվում օգտագործել **NULL** - ը:

Բաժին 3.5

Կառուցվածքներ (Structures)

Կառուցվածքային տիպն իրենից ներկայացնում է դաշտերի (փոփոխականների) կամ ֆունկցիաների համախումբ, որոնք կարող են լինել տարբեր տիպերի, ունենալ տարբեր երկարություններ, բայց որոնք համախմբված են մեկ հայտարարության մեջ: Դրա տեսքը հետևյալն է.

```
struct անուն {  
    տիպ1 տարր1;  
    տիպ2 տարր2;  
    տիպ3 տարր3;  
    .  
    .  
} օբյեկտի անուն;
```

որտեղ **անունը** կառուցվածքի տիպի անունն է, իսկ **օբյեկտի անունը**՝ տվյալ կառուցվածքային տիպի՝ մեկ կամ մի քանի օբյեկտների (փոփոխականների) անունները: Կառուցվածքային տիպի այսպիսի հայտարարման արդյունքում ստեղծվում է մի նոր տիպ՝ **անուն** անունով, ինչպես նաև այդ տիպի նոր օբյեկտ **օբյեկտի անուն** անունով: Եթե կառուցվածքային տիպի անունը չգրենք, կստեղծվի միայն **օբյեկտի անուն** անունով օբյեկտ, և հնարավոր չի լինի ստեղծել համապատասխան տիպի այլ օբյեկտներ: Կարելի է վարվել նաև հակառակ կերպ՝ նշել կառուցվածքային տիպի անուն և չնշել **օբյեկտի անունը**: Այս դեպքում կստեղծվի համապատասխան տիպը, իսկ օբյեկտ չի ստեղծվի: Օրինակ՝

```
struct apranq {  
    char anun [30];  
    float gin;  
};  
  
apranq xndzor;  
apranq dzmeruk, bal;
```

Այս օրինակում սկզբից հայտարարեցինք կառուցվածքային մոդել՝ **apranq**՝ երկու դաշտերով՝ **anun** և **gin**, որոնց տիպերը տարբեր են: Այնուհետև օգտագործեցինք կառուցվածքային տիպի անունը (**apranq**)՝ այդ տիպի երեք օբյեկտներ (**xndzor**, **dzmeruk** և **bal**) հայտարարելու համար: Ասվածը նշանակում է, որ **apranq** - ը դարձավ **int**, **char**, **short** և այլ ֆունդամենտալ տիպերի նման մի տիպ:

Վերևում ասվածից հետևում է, որ նախորդ օրինակի փոխարեն կարող էինք գրել:

```
struct apranq {  
    char anun [30];  
    float gin;  
} xndzor, dzmeruk, bal;
```

Այս դեպքում կառուցվածքի **անունը** կարելի էր անտեսել: Բայց այդ դեպքում այլևս չէինք կարողանա հայտարարել այդ տիպի նոր օբյեկտներ:

Հնարավոր է նաև միաժամանակ երկու (**անուն** և **օբյեկտի անուն**) պարամետրերի բաց թողնելը, սակայն այդ դեպքում, գրվածը կլինի բացարձակապես անիմաստ:

Շատ կարևոր է իրարից տարբերել **կառուցվածքային մոդելի** և **կառուցվածքային օբյեկտի** գաղափարները: Կառուցվածքային **մոդելը** տիպ է, իսկ **օբյեկտը**՝ համապատասխան տիպի օբյեկտ:

Կառուցվածքային տիպի օբյեկտներ հայտարարելուց հետո կարող ենք աշխատել դրանց դաշտերի հետ՝ դնելով կետ (.) սիմվոլը օբյեկտի անվան և դաշտի անվան միջև.

```
xndzor.anun
xndzor.gin
dzmeruk.anun
dzmeruk.gin
bal.anun
bal.gin
```

Այս արտահայտություններից երեքը՝ **xndzor.anun**, **dzmeruk.anun**, **bal.anun**, **char[30]** տիպի են, իսկ մյուս երեքը՝ **xndzor.gin**, **dzmeruk.gin**, **bal.gin**՝ **float** տիպի:

Դիտարկենք մեկ այլ օրինակ:

```
// karucvacqneri orinak
#include <iostream>
using namespace std;
#include <stdlib.h>

struct usanox {
    char anun [50];
    int xumb;
} usanox1, usanox2;

void tpel_usanoxin (usanox u);

int main ()
{
    char buffer [50];

    cout << "Mutqagrek dzer anun@: ";
    cin.getline (usanox1.anun,50);
    cout << "Mutqagrek dzer xumb@: ";
    cin.getline (buffer,50);
    usanox1.xumb = atoi(buffer);

    cout << "Mutqagrek dzer anun@: ";
    cin.getline (usanox2.anun,50);
    cout << "Mutqagrek dzer xumb@: ";
    cin.getline (buffer,50);
    usanox2.xumb = atoi(buffer);

    cout << endl;

    tpel_usanoxin (usanox1);
    tpel_usanoxin (usanox2);
```

```

return 0;
}

void tpel_usanoxin (usanox u)
{
    cout << "Anun: " << u.anun << endl;
    cout << "Xumb: " << u.xumb << "\n";
}

```

Mutqagrek dzer anun@: Petros
Mutqagrek dzer xumb@: 348
Mutqagrek dzer anun@: Poghos
Mutqagrek dzer xumb@: 111

Anun: Petros
Xumb: 348
Anun: Poghos
Xumb: 111

Այս օրինակը ցույց է տալիս, թե ինչպես ենք կարող օգտագործել կառուցվածքի տարրերը: Ուշադրություն դարձրենք նաև այն բանին, որ **tpel_usanoxin** ֆունկցիային որպես պարամետր փոխանցվում է **usanox** կառուցվածքային տիպի փոփոխական, այլ ոչ կառուցվածքային օբյեկտի դաշտերն առանձին-առանձին:

Ստորև բերված է կառուցվածքների մեկ այլ օրինակ, որտեղ զանգված է օգտագործվում:

```

// karucvacqneri orinak
#include <iostream>
using namespace std;
#include <stdlib.h>

#define USANOXNERI_QANAK 5

struct usanox {
    char anun [50];
    int xumb;
} usanoxner[USANOXNERI_QANAK];

void tpel_usanoxin (usanox u);

int main ()
{
    char buffer [50];
    int n;

    for (n=0; n<USANOXNERI_QANAK; n++)
    {
        cout << "Mutqagrek dzer anun@: ";
        cin.getline (usanoxner[n].anun,50);
        cout << "Mutqagrek dzer xumb@: ";
        cin.getline (buffer,50);
        usanoxner[n].xumb = atoi(buffer);
    }

    cout << endl;
}

```



```
for (n=0; n<USANOXNERI_QANAK; n++)
    tpel_usanoxin (usanoxner[n]);

return 0;
}

void tpel_usanoxin (usanox u)
{
    cout << "Anun: " << u.anun << endl;
    cout << "Xumb: " << u.xumb << "\n";
}

Mutqagrek dzer anun@: aaa
Mutqagrek dzer xumb@: 111
Mutqagrek dzer anun@: bbb
Mutqagrek dzer xumb@: 222
Mutqagrek dzer anun@: ccc
Mutqagrek dzer xumb@: 333
Mutqagrek dzer anun@: ddd
Mutqagrek dzer xumb@: 444
Mutqagrek dzer anun@: eee
Mutqagrek dzer xumb@: 555

Anun: aaa
Xumb: 111
Anun: bbb
Xumb: 222
Anun: ccc
Xumb: 333
Anun: ddd
Xumb: 444
Anun: eee
Xumb: 555
```

Ցուցիչներ կառուցվածքներին

Ինչպես բոլոր այլ տիպերը, կառուցվածքները ևս կարող են ցուցադրված լինել ցուցիչներով: Կանոնները նույնն են, ինչ ցանկացած ֆունդամենտալ տիպի համար. ցուցիչը պետք է հայտարարված լինի՝ որպես ցուցիչ կառուցվածքին.

```
struct usanox {
    char anun [50];
    int xumb;
};
```

```
usanox u;
usanox * pu;
```

Այստեղ, **u** - ն **usanox** կառուցվածքային տիպի օբյեկտ է, իսկ **pu** - ն՝ ցուցիչ, որը ցույց է տալիս **usanox** կառուցվածքային տիպի օբյեկտների: Այսպիսով, հետևյալ արտահայտությունը ևս ճիշտ կլինի.

```
pu = &u;
```

Օրինակ՝

```

#include <iostream>
using namespace std;
#include <stdlib.h>

struct usanox {
    char anun [50];
    int xumb;
};

int main ()
{
    char buffer[50];

    usanox u;
    usanox * pu;
    pu = &u;

    cout << "Mutqagrek dzer anun@: ";
    cin.getline (pu->anun,50);
    cout << "Mutqagrek dzer xumb@: ";
    cin.getline (buffer,50);
    pu ->xumb = atoi (buffer);

    cout << "\nDuc mutqagrecik:\n";
    cout << "Anun: " << pu->anun << endl;
    cout << "Xumb: " << pu->xumb << endl;

    return 0;
}

```

```

Mutqagrek dzer anun@: Hovik
Mutqagrek dzer xumb@: 249

```

```

Duc mutqagrecik:
Anun: Hovik
Xumb: 249

```

Այս ծրագիրն օգտագործում է մի նոր գաղափար՝ -> օպերատորը: Սա հետհասցեավորման օպերատոր է, որն օգտագործվում է միայն կառուցվածքների կամ կլասների ցույց տվող ցուցիչների հետ: Այս օպերատորը թույլ է տալիս չօգտագործել * սիմվոլը՝ կառուցվածքի դաշտին դիմելիս:

pu->anun

համարժեք է հետևյալին՝

(*pu).anun

Պետք է տարբերել **(*pu).anun** - ը ***pu.anun** արտահայտությունից: ***pu.anun** - ը համարժեք է ***(pu.anun)** արտահայտությանը, որը **pu** կառուցվածքի **anun** դաշտի ցույց տված արժեքն է (մինչդեռ **anun** - ը մեր օրինակում ցուցիչ չէ):

Հետևյալ ցուցակն ամփոփում է ցուցիչների և կառուցվածքների հնարավոր համակցումները.

Արտահայտություն	Բացատրություն	Համարժեք
<code>pu.anun</code>	<code>pu</code> կառուցվածքի <code>anun</code> դաշտ	
<code>pu->anun</code>	<code>pu</code> - ի ցույց տված կառուցվածքի <code>anun</code> դաշտ	<code>(*pu).anun</code>
<code>*pu.anun</code>	<code>pu</code> կառուցվածքի <code>anun</code> դաշտի ցույց տված արժեք	<code>*(pu.anun)</code>

Կառուցվածքների խտացում

Որպես կառուցվածքի տարր (դաշտ) կարող են նաև հանդես գալ այլ կառուցվածքների օբյեկտներ.

```
struct usanox {
    char anun [50];
    int xumb;
}

struct hosq {
    int hamar;
    usanox usanoxner[50];
} arajin_hosq, erkrord_hosq;

hosq * phosq = &arajin_hosq;
```

Այս հայտարարություններից հետո կարող ենք օգտագործել հետևյալ արտահայտությունները`

```
arajin_hosq.hamar
arajin_hosq.usanoxner[i].anun
arajin_hosq.usanoxner[i].xumb
erkrord_hosq.hamar
erkrord_hosq.usanoxner[i].anun
erkrord_hosq.usanoxner[i].xumb
phosq->hamar
phosq->usanoxner[i].anun
phosq->usanoxner[i].xumb
```

որտեղ `i` - ն `0 - 49` – ը ինչ-որ թիվ է:

Այս բաժնում ներկայացված կառուցվածքների հնարավորությունները նույնն են, ինչ `C` լեզվում: Սակայն `C++` լեզվում կառուցվածքների հնարավորություններն ավելի են ընդլայնվել և հասցվել կլասների մակարդակին, միայն այն տարբերությամբ, որ դրանց տարրերը համարվում են **public**: Այս ամենին կծանոթանանք **4.1** բաժնում:

Բաժին 3.6

Օգտագործողի հայտարարած տիպեր

Մենք արդեն ծանոթ ենք ծրագրավորողի կողմից հայտարարվող տիպերի մի խմբի: Դրանք կառուցվածքներն են: Բայց դրանցից բացի, կան ծրագրավորողի կողմից հայտարարվող տիպերի այլ խմբեր:

Սեփական տիպերի հայտարարում (typedef).

C++ - ը հնարավորություն է տալիս սահմանել մեր սեփական տիպերը՝ հիմնվելով արդեն գոյություն ունեցող տիպերի վրա: Դա անելու համար պարզապես պետք է օգտագործել **typedef** բանալի-բառը: Գրելաձևը հետևյալն է՝

```
typedef գոյություն_ունեցող_տիպ նոր_տիպի_անուն;
```

որտեղ **գոյություն_ունեցող_տիպը** որևէ գոյություն ունեցող տիպ է (C++ - ի ստանդարտ տիպերից կամ մեր սահմանած տիպ), իսկ **նոր_տիպի_անունը** այն անունն է, որը կստանա մեր հայտարարած նոր տիպը: Օրինակ՝

```
typedef char C;  
typedef unsigned int WORD;  
typedef char * tox;  
typedef char dasht [50];
```

Այստեղ հայտարարեցինք 4 նոր տիպեր՝ **C**, **WORD**, **tox** և **dasht**, որպես համապատասխանաբար **char**, **unsigned int**, **char*** և **char[50]**: Հայտարարելուց հետո կարող ենք դրանք օգտագործել ինչպես ցանկացած սովորական տիպ:

```
C misimvol, miurishsimvol, *ptsimvoll1;  
WORD myword;  
tox ptsimvol2;  
dasht anun;
```

typedef - ը կարող է օգտագործվել, եթե, օրինակ, ծրագրում օգտագործվում է մի տիպ, որը կարող է անհրաժեշտ լինի փոխել մեկ այլ տիպով ծրագրի հաջորդ տարբերակում, կամ եթե օգտագործվող տիպը շատ երկար անուն ունի, և հարմար է նրա անունը փոխարինել մեկ ուրիշ՝ ավելի կարճ նույնարկիչով:

Միավորումներ (Unions)

Միավորումները հնարավորություն են տալիս հիշողության որոշակի տիրույթի դիմել որպես տարբեր տիպեր: Միավորման բոլոր տարրերը պահվում են հիշողության միևնույն հասցեից սկսած: Միավորումներ հայտարարելու գրելաձևը ճիշտ նույնն է, ինչ կառուցվածքների, սակայն միավորումներն ունեն բոլորովին այլ հատկություններ:

```

union անուն {
    տիպ1 էլէմենտ1;
    տիպ2 էլէմենտ2;
    տիպ3 էլէմենտ3;
    .
    .
} օբյեկտի_անուն;

```

Քանզի միավորման բոլոր տարրերը պահվում են հիշողության միևնույն հասցեից սկսած, միավորման չափը համընկնում է նրա ամենամեծ տարրի չափի հետ: Օրինակ՝

```

union imtipper_tip {
    char c;
    int i;
    float f;
} iptiper;

```

Հայտարարում է 3 տարր՝

```

imtipper.c
imtipper.i
imtipper.f

```

բոլորը՝ տարրեր տեսակների: Քանի որ նրանք բոլորը պահվում են հիշողության մեջ նույն հասցեում, դրանցից մեկի վրա կատարված փոփոխությունը կազդի մյուսների վրա:

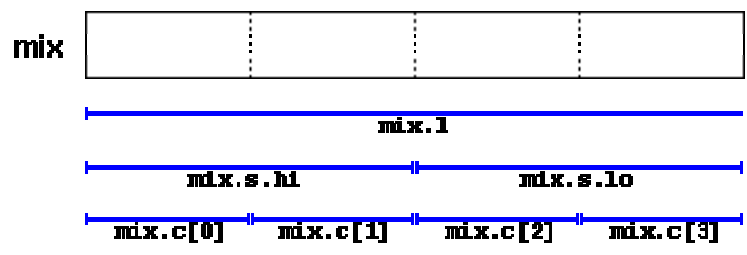
Ահա մեկ օրինակ՝

```

union mix_t {
    long l;
    struct {
        short hi;
        short lo;
    } s;
    char c[4];
} mix;

```

Հայտարարում է 3 անուններ՝ **mix.l**, **mix.s** և **mix.c**, որոնց միջոցով կարող ենք դիմել հիշողության միևնույն 4 բայթերին: Հիշողության այդ տիրույթին կարող ենք դիմել որպես **long**, **short** կամ **char**: Այս միավորումը կարելի է պատկերել հետևյալ գծագրի միջոցով՝



Անանուն (Anonymous) միավորումներ

Եթե C++ - ում կառուցվածքի մեջ տեղադրենք միավորում՝ առանց այդ միավորման համար օբյեկտ նշելու (ձևավոր փակագծերից հետո), ապա այդ միավորումը կլինի անանուն, և կկարողանանք անմիջականորեն դիմել նրա տարրերին՝ իրենց անուններով: Օրինակ՝

<u>միավորում</u>	<u>անանուն միավորում</u>
<pre>struct { char vernagir[50]; char hexinak[50]; union { float dollar; int dram; } gin; } girq;</pre>	<pre>struct { char vernagir[50]; char hexinak[50]; union { float dollar; int dram; }; } girq;</pre>

Այս օրինակների միակ տարբերությունն այն է, որ առաջինում նշեցինք գինը պարունակող օբյեկտ՝ **gin**, իսկ մյուսում՝ ոչ: Տարբերությունը նրանում է, որ **dollar** և **dram** անդամներին դիմելիս առաջին օրինակում պետք է գրենք

```
girq.gin.dollar
girq.gin.dram
```

իսկ երկրորդում՝

```
girq.dollar
girq.dram
```

Կրկին հիշենք, որ միավորման մեջ գտնվող **dollar** և **dram** անդամները գտնվում են հիշողության նույն հասցեում, հետևաբար դրանք չեն կարող օգտագործվել 2 տարբեր արժեքներ պահելու համար: Սա նշանակում է, որ գինը պետք է լինի կամ դուլարով, կամ դրամով:

Թվարկումներ (Enumerations) (enum)

Թվարկումներն օգտագործվում են, այնպիսի տիպեր ստեղծելու համար, որոնց արժեքները չեն սահմանափակվում թվային կամ սիմվոլային հաստատուններով և ոչ էլ **true** և **false** - ով: Գրելաձևը հետևյալն է՝

```
enum անուն {
    արժեք1,
    արժեք2,
    արժեք3,
    .
    .
} օբյեկտի_անուն;
```

Թվարկումների կիրառությունները և աշխատանքը հասկանալու համար դիմենք օրինակի օգնությանը: Ստեղծենք **guyn** անունով մի նոր տիպ, որում կարող ենք պահել տարբեր գույներ.

```
enum guyn {sev, kapuyt, kanach, yerknaguyn, karmir, dixin, spitak};
```

Ուշադրություն դարձնենք, որ օրինակը չի պարունակում որևէ գոյություն ունեցող տիպ. այն ոչնչի վրա հիմնված չէ: **guyn** տիպը հիմա գոյություն ունեցող տիպ է, որի հնարավոր արժեքներն այն գույներն են, որոնք գրված են {} փակագծերի միջև: Օրինակ՝

```
guyn im_guyn;  
  
im_guyn = kapuyt;  
if (im_guyn == kanach) im_guyn = karmir;
```

Իրականում մեր թվարկած արժեքները դիտվում են որպես ամբողջ թվեր: Եթե ոչ մի թիվ չենք նշում, ապա թարգմանիչն անդամներին համարակալում է՝ սկսելով **0** - ից: Նախորդ օրինակում **guyn** թվարկման մեջ **sev** - ը կհամապատասխանի **0** - ին, **kapuyt** - ը՝ **1** - ին, **kanach** - ը՝ **2** - ին, և այսպես շարունակ:

Եթե մեր թվարկման հնարավոր արժեքներից մեկին տանք որևէ ամբողջ թիվ, ապա հաջորդող արժեքները կհամարակալվեն՝ սկսած տրված համարից: Օրինակ՝

```
enum amisner { hunvar=1, petrval, mart, april,  
               mayis, hunis, hulis, ogostos,  
               september, hoktember, noyember, dektember } y2k;
```

Այստեղ **y2k** փոփոխականը, որը **amisner** տիպի է, կարող է պարունակել հնարավոր 12 արժեքներից յուրաքանչյուրը՝ **hunvar** - ով սկսած, **dektember** - ով վերջացրած. սրանք համարժեք են **1** - ից **12** թվերին, այլ ոչ **0** - ից **11**, քանի որ **hunvar** - ին տվել էինք **1** արժեքը:

Բաժին 4.1

Դասեր (Classes)

Դասերն օգտագործվում են տվյալներն ու ֆունկցիաները մեկ միասնական համակարգի մեջ համախմբելու համար: Դասերը հայտարարվում են **class** բանալի-բառով.

```
class անուն {
    տեսանելիության_նշիչ_1:
        անդամ1;
    տեսանելիության_նշիչ_2:
        անդամ2;
    ...
} օբյեկտի_անուն;
```

որտեղ **անունը** դասի անունն է (պարտադիր չէ), իսկ **օբյեկտի_անունը**՝ տվյալ դասային տիպի մեկ կամ մի քանի օբյեկտների (փոփոխականների) անուններ (սա նույնպես կարելի է բաց թողնել): Դասի մարմինը կարող է պարունակել **անդամներ**, որոնք կարող են լինել կամ արժեքներ (փոփոխականներ, հաստատուններ), կամ ֆունկցիաներ: **Տեսանելիության նշիչները** կարող են լինել՝ **private**, **public** կամ **protected**: Դրանք ազդում են անդամների վրա հետևյալ կերպ.

- Դասի **private** անդամներին կարող են դիմել միայն այդ դասի մյուս անդամները կամ **բարեկամ** դասերը:
- **protected** անդամներին կարող են դիմել միայն այդ դասի մյուս անդամները, **բարեկամ** դասերը կամ էլ ժառանգող դասի անդամները :
- **public** անդամներին կարող ենք դիմել այդ դասի տեսանելիության ողջ տիրույթից:

Բարեկամ դասերն ու ֆունկցիաներն, ինչպես նաև դասերի ժառանգականությունը կուսումնասիրենք այս բաժնի հաջորդ գլուխներում:

Եթե հայտարարենք դասի անդամներ, առանց որևէ **տեսանելիության նշիչ** նշելու, ապա գրված անդամները կհամարվեն **private**:

Բերենք դասի մի օրինակ՝

```
class CRectangle {
    int x, y;
    public:
        void set_values (int,int);
        int area (void);
} rect;
```

Այս կողմը հայտարարում է **CRectangle** դասային տիպ և այդ տիպի **rect** անունով օբյեկտ: Այս դասը պարունակում է չորս անդամներ՝ երկու **int** տիպի փոփոխականներ (**x** և **y**)՝ հայտարարված **private** տեսանելիության տիրույթում, և երկու՝ (**set_values()** և **area()**) ֆունկցիաներ (միայն նախատիպերը)՝ հայտարարված **public** տեսանելիության տիրույթում:

Ուշադրություն դարձնենք նաև այն բանին, որ այստեղ որպես տիպ հանդես է գալիս **CRectangle** - ը, իսկ որպես այդ տիպի օբյեկտ՝ **rect** - ը:

Ծրագիր գրելիս կարող ենք օգտագործել մեր հայտարարած օբյեկտի՝ **public** տեսանելիության տիրույթում գտնվող անդամները հետևյալ կերպ.

```
օբյեկտի_անուն.անդամ
```

Այսինքն դասի անդամներին դիմում ենք ճիշտ այնպես, ինչպես կդիմեինք կառուցվածքի նդամներին: Օրինակ՝

```
rect.set_value (3,4);  
myarea = rect.area();
```

Սակայն չենք կարող անմիջականորեն օգտագործել **CRectangle** դասի **x** և **y** փոփոխականները, քանի որ դրանք գտնվում են դասի **private** տեսանելիության տիրույթում և տեսանելի են միայն դասի սահմաններում, այսինքն՝ միայն դասի անդամների համար:

Դիտարկենք ևս մի օրինակ՝

```
// daseri orinak  
#include <iostream>  
using namespace std;  
  
class CRectangle {  
    int x, y;  
public:  
    void set_values (int,int);  
    int area (void) {return (x*y);}  
};  
  
void CRectangle::set_values (int a, int b) {  
    x = a;  
    y = b;  
}  
  
int main () {  
    CRectangle rect;  
    rect.set_values (3,4);  
    cout << "area: " << rect.area();  
}
```

```
area: 12
```

Այս ծրագրի մեջ նորությունը :: օպերատորն է: Այն օգտագործվում է դասի հայտարարությունից դուրս դասի անդամ որոշելու համար: Ուշադրություն դարձնենք, որ **area()** անդամ ֆունկցիան հայտարարեցինք և որոշեցինք դասի մեջ, իսկ **set_values()** անդամ ֆունկցիան նկարագրեցինք դասի մեջ և որոշեցինք դասից դուրս:

Տեսանելիության տիրույթի :: օպերատորը ցույց է տալիս, թե տվյալ անդամ-ֆունկցիան, որ դասին է պատկանում, և ապահովում է բոլոր այն

հատկությունները, որոնք ներկա կլինեն, եթե այդ անդամ-ֆունկցիան որոշված լինե՞ր տվյալ դասի մեջ: Նկատենք, որ `set_values()` ֆունկցիայում օգտագործեցինք `x` և `y` փոփոխականները, որոնք գտնվում են `CRectangle` դասի `private` տեսանելիության տիրույթում:

Անդամ-ֆունկցիան դասի մեջ կամ դասից դուրս որոշելու միակ տարբերությունն այն է, որ երբ որոշվում է դասի մեջ, այն թարգմանիչի կողմից համարվում է `inline` (տողամիջյան) ֆունկցիա, իսկ երբ որոշվում է դասից դուրս, համարվում է դասի սովորական անդամ-ֆունկցիա:

Այս օրինակում դասի `x` և `y` անդամները հայտարարեցինք `private` և որոշեցինք `set_values()` ֆունկցիան, որը արժեքներ է վերագրում այդ անդամներին: Մեր դեպքում դասի որևէ օբյեկտի `x` և `y` անդամները չեն կարող անմիջականորեն փոփոխվել: Փոփոխությունը հնարավոր է միայն `set_values()` միջնորդ ֆունկցիայի միջոցով: Մեր փոքր օրինակում, իհարկե, այս մոտեցման իմաստը այդքան էլ հասկանալի չէ, սակայն ավելի մեծ նախագծերում սա լայնորեն կիրառվող մոտեցում է: Դիտարկենք հետևյալ օրինակը: Դիցուք որևէ դասի մեջ սահմանել ենք `float` տիպի `tokos` փոփոխականը: Տրամաբանական է ենթադրել, որ `tokos` – ը պիտի լինի 0 – 100 միջակայքի թիվ: Այդ սահմանափակումը ստուգելու համար `tokos` – ը կհայտարարենք `private` և կորոշենք `bool set_tokos(float)` անդամ-ֆունկցիան (`private` տիրույթում), որը ընդունելով `float` տիպի արգումենտ կհամոզվի, որ այն համապատասխան միջակայքի է և կկատարի անհրաժեշտ վերագրումը:

```
#include <iostream>
using namespace std;
#include <string.h>

class Baxadrich {
    char *anun;
    float tokos;
public:
    void set_tokos (float);
    void set_anun(char *);
    void tpel();
};

void Baxadrich::set_tokos(float newtokos) {
    if(newtokos >= 0 && newtokos <= 100) {
        tokos = newtokos;
    } else {
        cout << "tokos-@ piti lini 0-ic 100" << endl;
    }
}

void Baxadrich::set_anun(char *nor_anun) {
    anun = new char[strlen(nor_anun) + 1];
    strcpy(anun, nor_anun);
}

void Baxadrich::tpel() {
    cout << "Anun: " << anun << endl;
    cout << "Tokos: " << tokos << endl;
}
```

```
int main () {
    Baxadrich bax_1;
    bax_1.set_anun("Spirt");
    bax_1.set_tokos(40.1);
    bax_1.tpel();
    bax_1.set_anun("Jur");
    bax_1.set_tokos(140.1);
    bax_1.tpel();
}
```

```
Anun: Spirt
Tokos: 40.1
Anun: Jur
Tokos: 40.1
```

Դասերի առավելություններից մեկն էլ այն է, որ կարող ենք հայտարարել տրված դասի մեկից ավել օբյեկտներ: Օրինակ՝ **CRectangle** դասի օրինակում ավելացնենք այդ տիպի ևս մի օբյեկտ՝ **rectb**:

```
// klasi orinak
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle rect, rectb;
    rect.set_values (3,4);
    rectb.set_values (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

Ուշադրություն դարձնենք այն բանին, որ **rect.area()** ֆունկցիայի կանչը նույն արդյունքը չի տա, ինչ **rectb.area()** ֆունկցիային կանչը: Դրա պատճառն այն է, որ այս երկու՝ **rect** և **rectb** օբյեկտներից յուրաքանչյուրն ունի իր սեփական **x** և **y** փոփոխականները ու իր սեփական **set_value()** և **area()** ֆունկցիաները:

Սրա վրա են հիմնված **օբյեկտի** և **օբյեկտակողմնորոշված** ծրագրավորման գաղափարները: Դրանցում փոփոխականները և ֆունկցիաներն օբյեկտի հատկություններն են՝ ի տարբերություն կառուցվածքային ծրագրավորման, որտեղ օբյեկտները ներկայանում են որպես ֆունկցիայի պարամետրեր: Այս և հաջորդ բաժիններում կուսումնասիրենք այս գաղափարի առավելությունները:

Կառուցիչներ և փլուզիչներ (Constructors and Destructors)

Սովորաբար օբյեկտի ստեղծման ժամանակ անհրաժեշտ է լինում սկզբնարժեքավորել փոփոխականներ կամ էլ հատկացնել դիմանիկ հիշողություն կանխելու համար անորոշ արժեքների վերադարձի հնարավորությունը: Օրինակ՝ ինչ կկատարվի նախորդ ծրագրում, եթե կանչենք `area()` ֆունկցիան՝ առանց նախորոք կանչելու `set_values()` - ը: Չնայած նրան, որ `x` և `y` փոփոխականներին արժեք չենք վերագրել՝ հիշողության այն տիրույթը, որ տրվել է այդ փոփոխականներին, կպարունակի որոշ արժեքներ, որոնք գրվել են ինչ-որ այլ ծրագրի կողմից կամ հայտնվել են այնտեղ դեռևս համակարգիչը միացնելիս: Կոնկրետ դեպքում այդ արժեքները մեր ծրագրի կողմից կդիտվեն որպես ամբողջ թվեր (`int`), և `area()` ֆունկցիայի կատարման արդյունքում կվերադարձվի այդ երկու (մեզ համար անորոշ) թվերի արտադրյալը, որը կարող է լինել նաև բացասական:

Սա կանխելու համար դասը կարող է պարունակել հատուկ ֆունկցիա՝ **կառուցիչ**: **Կառուցիչ** ֆունկցիա հայտարարելու համար պարզապես պետք է անվանել անդամ-ֆունկցիան դասի անունով: Այս ֆունկցիան ավտոմատ կանչվելու է տվյալ դասային տիպի նոր օբյեկտներ ստեղծելիս:

Օրինակ.

```
// klaseri orinak
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
public:
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

Նկատենք, որ այս ծրագրի արդյունքը նույնն է, ինչ նախորդինը: Այս օրինակում պարզապես փոխարինեցինք `set_values()` անդամ-ֆունկցիան դասի կառուցիչով: Ուշադրություն դարձնենք, թե ինչպես են պարամետրները փոխանցվում կառուցիչին՝ նոր օբյեկտների ստեղծման ժամանակ.

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Նմատենք նաև այն փաստը, որ ոչ դասում՝ կառուցիչի նկարագրման մեջ, և ոչ էլ կառուցիչի որոշման մեջ գրված չէ վերադարձվող արժեքի տիպը (նույնիսկ **void** չի գրված): Կառուցիչը երբեք արժեք չի վերադարձնում, նույնիսկ **void** տիպի:

Փլուզիչը կատարում է հակառակ գործողությունը: Այն ավտոմատ կանչվում է, երբ օբյեկտը ազատվում (ջնջվում) է հիշողությունից: Դա կամ օբյեկտի կյանքի տևողության ավարտի հետևանք է (օրինակ, եթե օբյեկտը հայտարարված էր որպես լոկալ օբյեկտ ֆունկցիայի մեջ, և ֆունկցիան ավարտվեց), կամ էլ, եթե այն դինամիկ վերագրված օբյեկտ է և ազատվում է հիշողությունից՝ **delete** օպերատորի միջոցով:

Փլուզիչը պետք է ունենա նույն անունը, ինչ դասը, իսկ անունից առաջ դրված է ~ սիմվոլը: Փլուզիչը նույնպես պետք է ոչ մի արժեք չվերադարձնի:

Փլուզիչները հաճախ օգտագործվում են այն դեպքերում, երբ օբյեկտը ծրագրի ընթացքում պահանջել է դինամիկ հիշողություն, և որն էլ այժմ պիտի ազատվի:

Օրինակ.

```
// karucichneri ev pluzichneri orinak
#include <iostream>
using namespace std;

class CRectangle {
    int *width, *height;
public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
    width = new int;
    height = new int;
    *width = a;
    *height = b;
}

CRectangle::~CRectangle () {
    delete width;
    delete height;
}

int main () {
    CRectangle rect (3,4), rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

```
rect area: 12
rectb area: 30
```

Կառուցիչների գերբեռնում (Overloading Constructors)

Ինչպես ցանկացած այլ ֆունկցիա, կառուցիչը կարող է գերբեռնվել մի քանի ֆունկցիաներով, որոնք ունեն նույն անունը, բայց տարբեր քանակի և տիպի պարամետրեր:

Այն դեպքում, երբ հայտարարենք դաս, բայց չորոշենք որևէ կառուցիչ, թարգմանիչն ավտոմատ կստեղծի երկու գերբեռնված կառուցիչները՝ **դատարկ կառուցիչը** և **պատճենման կառուցիչը**: Օրինակ՝ հետևյալ դասի համար՝

```
class COrinak {
public:
    int a,b,c;
    void bazmapatkel (int n, int m) { a=n; b=m; c=a*b; };
};
```

որը չունի որևէ կառուցիչ, թարգմանիչն ավտոմատ կստեղծի հետևյալ կառուցիչները.

- **Դատարկ կառուցիչ**

Սա պարամետրեր չունեցող կառուցիչ է, որի մարմինը դատարկ է.

```
COrinak::COrinak () { };
```

- **Պատճենման կառուցիչ**

Սա միևնույն դասային տիպի մեկ պարամետրով կառուցիչ է, որը տրված օբյեկտի ցանկացած **ոչ ստատիկ** անդամ փոփոխականին վերագրում է փոխանցած օբյեկտի տվյալ անդամի պատճենը:

```
COrinak::COrinak (const COrinak& rv) {
    a=rv.a; b=rv.b; c=rv.c;
}
```

Բայց հիշենք, որ **դատարկ կառուցիչը** և **պատճենման կառուցիչն** օգտագործվում են միայն այն ժամանակ, երբ տվյալ դասի համար ոչ մի այլ կառուցիչ չի հայտարարված: Այն դեպքում, երբ հայտարարված է նույնիսկ մեկ կառուցիչ, այս երկուսից ոչ մեկը չի ստեղծվի:

Օրինակ.

```
// klasi karucichneri canrabernum
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};
```

```

CRectangle::CRectangle () {
    width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}

```

```

rect area: 12
rectb area: 25

```

Այս օրինակում **rectb** - ն հայտարարված է առանց պարամետրերի, այդ պատճառով այն սկզբնաբեքավորվում է այն կառուցիչով, որը ոչ մի պարամետր չի ընդունում և **width** և **height** փոփոխականներին վերագրում է **5** արժեքը:

Ուշադրություն դարձնենք նաև այն բանին, որ եթե հայտարարում ենք նոր օբյեկտ և չենք ուզում կառուցիչին փոխանցել որևէ պարամետր, ապա պետք չէ օգտագործել փակագծեր.

```

CRectangle rectb; // chisht
CRectangle rectb(); // sxal!

```

Ցուցիչներ դասերին

Ինչպես բոլոր մյուս տիպերը, դասերը ևս կարող են ցուցադրված լինել ցուցիչներով: Դասերին ցույց տվող ցուցիչներ հայտարարելու ձևը հետևյալն է.

*դասի անուն * ցուցիչի անուն;*

Օրինակ՝

```

CRectangle * prect;

```

ցուցիչ է **CRectangle** դասի օբյեկտի վրա:

Ինչպես վարվում էինք կառուցվածքների դեպքում, այնպես էլ այստեղ, դասերի դեպքում, որպեսզի դիմենք ցուցիչի ցույց տված դասի օբյեկտի անդամին, պետք է օգտագործենք -> օպերատորը: Օրինակ՝

```

// klasneri vra cucichner
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
}

```

```

public:
    void set_values (int, int);
    int area (void) {return (width * height);}
};

void CRectangle::set_values (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle a, *b, *c;
    CRectangle * d = new CRectangle[2];
    b= new CRectangle;
    c= &a;
    a.set_values (1,2);
    b->set_values (3,4);
    d->set_values (5,6);
    d[1].set_values (7,8);
    cout << "a area: " << a.area() << endl;
    cout << "*b area: " << b->area() << endl;
    cout << "*c area: " << c->area() << endl;
    cout << "d[0] area: " << d[0].area() << endl;
    cout << "d[1] area: " << d[1].area() << endl;
    return 0;
}

```

```

a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56

```

Ստորև ասվում է, թե ինչպես ենք կարող կարդալ ցուցիչների և դասերի օպերատորները (*, &, ., ->, []), որոնք հանդիպում են օրինակում.

- *x** նշանակում է **x** - ի ցույց տված օբյեկտ
- &x** նշանակում է **x** - ի հասցե
- x.y** նշանակում է **x** օբյեկտի **y** անդամ
- (*x).y** նշանակում է **x** - ի ցույց տված օբյեկտի **y** անդամ
- x->y** նշանակում է **x** - ի ցույց տված օբյեկտի **y** անդամ
- x[0]** նշանակում է **x** - ի ցույց տված առաջին օբյեկտ
- x[1]** նշանակում է **x** - ի ցույց տված երկրորդ օբյեկտ
- x[n]** նշանակում է **x** - ի ցույց տված (n+1) - րդ օբյեկտ

struct բանալի-բառով որոշված դասեր

C++ լեզվում կառուցվածքը (**struct**) ունի նույն հնարավորությունները, ինչ դասը (**class**), միայն այն տարբերությամբ, որ կառուցվածքի անդամները լռությամբ **public** են համարվում, միջդեռ դասինը՝ **private**: Ընդունված է կառուցվածքներն օգտագործել միայն տվյալներ պահելու համար, իսկ դասերը՝ և տվյալներ, և ֆունկցիաներ: Դասի անդամ-ֆունկցիաներին ընդունված է անվանել **մեթոդ**:

Բաժին 4.2

Օպերատորների գերբեռնում (Overloading Operators)

C++ - ը հնարավորություն է տալիս գերբեռնելու լեզվի ստանդարտ օպերատորները՝ սահմանելով տարբեր գործողություններ դասերի համար: Օրինակ՝

```
int a, b, c;  
a = b + c;
```

Ճիշտ է, քանի որ int տիպերի համար գումարման գործողությունն արդեն սահմանված է: Սակայն չի կարելի գրել

```
struct { char product [50]; float price; } a, b, c;  
a = b + c;
```

առանց սահմանելու համապատասխան գումարման գործողությունը: Միակ գործողությունը, որը մեքենայաբար սահմանվում է բոլոր դասերի և կառուցվածքների համար, վերագրման գործողությունն է, որը թույլ է տալիս որևէ դասային (կառուցվածքային) տիպի մի օբյեկտին վերագրել մինևույն տիպի մեկ այլ օբյեկտ:

C++ - ում կարող ենք սահմանել այն գործողությունները (օպերատորները), որոնք դեռևս որոշված չեն տվյալ դասի համար, և նույնիսկ կարող ենք ձևափոխել արդեն որոշված օպերատորները: Ստորև բերված է օպերատորների ցանկը, որոնք կարող են գերբեռնվել:

Ունար օպերատորներ.

!	Տրամաբանական ՈՉ
&	Հասցեավորում
~	Լրացում
*	Հետհասցեավորում
+	Ունար գումարում
++	Մեծացում
-	Ունար հանում
--	Փոքրացում
<i>Ձևափոխության օպերատորներ</i>	<i>Ձևափոխության օպերատորներ</i>

Բինար օպերատորներ.

,	Ստորակետ
!=	Անհավասարություն
%	Մոդուլ
%=	Մոդուլ / վերագրում
&	Բիթային ԵՎ
&&	Տրամաբանական ԵՎ
&=	Բիթային ԵՎ / վերագրում

*	Բազմապատկում
*=	Բազմապատկում / վերագրում
+	Գումարում
+=	Գումարում / վերագրում
-	Հանում
-=	Հանում / վերագրում
->	Անդամի ընտրում
->*	Ցուցիչ անդամի ընտրման վրա
/	Բաժանում
/=	Բաժանում / վերագրում
<	Փոքր
<<	Չախ տեղաշարժ
<<=	Չախ տեղաշարժ / վերագրում
<=	Փոքր կամ հավասար
=	Վերագրում
==	Հավասար
>	Մեծ
>=	Մեծ կամ հավասար
>>	Աջ տեղաշարժ
>>=	Աջ տեղաշարժ / վերագրում
^	Բիթային բացառիկ ԿԱՄ
^=	Բիթային բացառիկ ԿԱՄ / վերագրում
	Բիթային ԿԱՄ
=	Բիթային ԿԱՄ / վերագրում
	Տրամաբանական ԿԱՄ

Այլ օպերատորներ.

()	Ֆունկցիայի կանչ
[]	Զանգվածի էլեմենտի դիմում
new	Օպերատոր new
delete	Օպերատոր delete

Օպերատոր գերբեռնելու համար անհրաժեշտ է գրել դասի անդամ-ֆունկցիա, որի անունը կազմված է **operator** բառից և դրան կցած օպերատորի նշանից, որը մենք ուզում ենք գերբեռնել: Տեսքը հետևյալն է՝

տիպ operator նշան (պարամետրեր);

Ունար օպերատորներ ծանրաբեռնելիս, անդամ ֆունկցիաները *պարամետրեր* չունեն (բացառություն են կազմում մեծացման և փոքրացման օպերատորների պոստֆիքսային տեսքերը՝ ++, --), իսկ բինար օպերատորների դեպքում՝ ունեն ճիշտ մեկ *պարամետր*, որի տիպը պետք է լինի հենց այդ դասի, կամ ժառանգվող դասի (կանցնենք մի փոքր ուշ) տիպը:

Ստորև բերված է մի օրինակ, որում գերբեռնվում է + օպերատորը՝ երկու երկչափ վեկտորներ գումարելու համար: Երկու երկչափ վեկտորներ գումարել նշանակում

Է գումարել դրանց համապատասխան օպերատորները: Այսինքն այս դեպքում **a+b** վեկտորների գումարման արդյունքը պետք է լինի **(3+1,1+2) = (4,3)** երկչափ վեկտորը.

```
// vektorner: operatorneri canrabernman orinak
#include <iostream>
using namespace std;

class CVector {
public:
    int x,y;
    CVector () {x = 0; y = 0;};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
    x = a;
    y = b;
}

CVector CVector::operator+ (CVector param) {
    CVector temp;
    temp.x = x + param.x;
    temp.y = y + param.y;
    return (temp);
}

int main () {
    CVector a (3,1);
    CVector b (1,2);
    CVector c;
    c = a + b;
    cout << c.x << ", " << c.y;
    return 0;
}

4,3
```

CVector դասի **operator+** ֆունկցիան պատասխանատու է թվաբանական + օպերատորի գերբեռնման համար: Այն կարող է կանչվել հետևյալ երկու ձևերով`

```
c = a + b;
c = a.operator+ (b);
```

Ուշադրություն դարձրեք նաև այն բանին, որ այս օրինակում հայտարարել ենք կառուցիչ` հետևյալ մարմնով`

```
CVector () {x = 0; y = 0;};
```

Սա պարտադիր է, քանի որ այստեղ սահմանել ենք մեկ այլ կառուցիչ`

```
CVector (int, int);
```

և այդ պատճառով **լռության կառուցիչներից** ոչ մեկը չի ստեղծվի: Այս դեպքում ինքներս ստիպված կլինենք հայտարարել այդպիսի կառուցիչ, հակառակ դեպքում

հետևյալ արտահայտությունը կհամարվի սխալ՝

```
CVector c;
```

Լռությամբ որոշվող **վերագրման օպերատորը** պատճենում է պարամետր օբյեկտի (աջ մասում գրված օբյեկտի) բոլոր **ոչ ստատիկ** անդամների պարունակությունը ձախ մասում գրված նույն տիպի օբյեկտի մեջ: Բայց, իհարկե, կարող ենք վերասահմանել այն մեր կամեցածի պես:

Չնայած նրան, որ պարտադիր չէ, որ գերբեռնված գործողությունները համապատասխանեն մաթեմատիկական գործողություններին կամ դրանց իմաստին, սակայն խորհուրդ է տրվում պահպանել այս պայմանը: Օրինակ՝ ցանկալի չէ, որ + օպերատորը իրականացնի մաթեմատիկական տարբերություն գործողությունը:

Չնայած, որ **operator+** ֆունկցիայի նախատիպը կարող է թվալ ակնհայտ, քանի որ այն վերցնում է օպերատորի աջ մասը որպես **operator+** ֆունկցիայի պարամետր, սակայն մյուս օպերատորների դեպքը այնքան էլ ակնհայտ չէ: Ստորև բերված է ցուցակ, թե ինչպես պետք է հայտարարվեն մյուս օպերատորների նախատիպերը (փոխարինե՛ք @ սիմվոլը համապատասխան օպերատորով):

Արտահայտություն	Օպերատոր	Անդամ ֆունկցիա	Գլոբալ ֆունկցիա
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)
a@b	+ - * / % ^ & < > == != <= >= << >> && ,	A::operator@(B)	operator@(A, B)
a@b	= += -= *= /= %= ^= &= = <<= >>= []	A::operator@(B)	-
a(b, c...)	()	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

որտեղ **a** – ն **A** դասի օբյեկտ է, **b** – ն՝ **B** դասի, **c** – ն՝ **C** դասի:

Այս ցուցակում նշված է օպերատորներ գերբեռնելու երկու եղանակ՝ որպես դասի անդամ-ֆունկցիա կամ որպես գլոբալ ֆունկցիա: Երկրորդ՝ գլոբալ ֆունկցիայի դեպքը կուսումնասիրենք ավելի ուշ:

this բանալի-բառը

this բանալի-բառը դասի սահմաններում ցույց է տալիս տվյալ դասի օգտագործվող օբյեկտը:

Այն կարող է օգտագործվել, օրինակ, ստուգելու, թե արդյոք փոխանցված պարամետրը հենց ինքը՝ օբյեկտն է, թե ոչ: Օրինակ.

```
// this
#include <iostream>
using namespace std;

class CDummy {
public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
    if (&param == this) return 1;
    else return 0;
}

int main () {
    CDummy a;
    CDummy* b = &a;
    if ( b->isitme(a) )
        cout << "ayo, &a-n b-n e!";
    return 0;
}
```

ayo, &a-n b-n e!

this - ը նաև սովորաբար օգտագործվում է **operator=** անդամ-ֆունկցիաներում, որոնք վերադարձնում են օբյեկտներ՝ հղումով (&): Նախորդ վեկտորների օրինակում կարող էինք գրել **operator=** ֆունկցիան՝ հետևյալ կերպ.

```
CVector& CVector::operator= (const CVector& param)
{
    x=param.x;
    y=param.y;
    return *this;
}
```

Սա իրականում այն կողմն է, որը ավելացվում է մեր դասին մեքենայաբար (թարքմանիչի կողմից), եթե ինքներս չենք սահմանում **operator=** անդամ-ֆունկցիա: Օպերատորների նմանատիպ իրականացումը հնարավորություն է տալիս օգտագործելու մի գործողության արդյունքն որպես մյուսի արգումենտ: Օրինակ՝ **operator=** - ի այս որոշման դեպքում կարող ենք գրել **a = b = c**, որը կնշանակի **c** - ի արժեքը տալ **b** - ին, այնուհետև **b** - ի նոր արժեքը տալ **a** - ին:

Ունար օպերատորների ծանրաբեռնում

Ծանրաբեռնենք ++ օպերատորի պրեֆիքսային և պոստֆիքսային տեսքերը.

```
// vektorner: operatorneri canraberberman orinak
#include <iostream>
using namespace std;
```

```

class CVector {
public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
    CVector operator ++ (); //Prefixayin depq@
    CVector operator ++ (int); //Postfixayin depq@
};

CVector::CVector (int a, int b) {
    x = a;
    y = b;
}

CVector CVector::operator+ (CVector param) {
    CVector temp;
    temp.x = x + param.x;
    temp.y = y + param.y;
    return temp;
}

CVector CVector::operator++ () {
    x++;
    y++;
    return *this;
}

CVector CVector::operator++ (int) {
    CVector temp = *this;
    ++*this;
    return temp;
}

int main () {
    CVector a (3,1);
    CVector b (1,2);
    CVector c;
    c = a + b;
    cout << "c: (" << c.x << ", " << c.y << ")" << endl;
    cout << "c++: (" << (c++).x << ", " << c.y << ")" << endl;
    cout << "c: (" << c.x << ", " << c.y << ")" << endl;
    cout << "++c: (" << (++c).x << ", " << c.y << ")" << endl;
    cout << "c: (" << c.x << ", " << c.y << ")" << endl;
    return 0;
}

```

```

c: (4,3)
c++: (4,4)
c: (5,4)
++c: (6,5)
c: (6,5)

```

C++ -ում ++ օպերատորի երկու տեսքերն էլ նկարագրվում են մեկ օպերատորի՝ operator++ -ի միջոցով, սակայն, պրեֆիքսային դեպքում, այն, որպես պարամետր, ոչինչ չի ընդունում, իսկ մյուս՝ պոստֆիքսային դեպքում, ընդունում է int տիպի պարամետր:

Ինչպես տեսնում եք, պրեֆիքսային տեսքում մենք սկզբում փոփոխում ենք տվյալ դասի անդամները, այնուհետև վերադարձնում հենց ինքը՝ դասը, մինչդեռ պոստֆիքսային տեսքում սկզբում պահումք ենք տվյալ դասի պատճեն, այնուհետև փոփոխում դասի անդամները և վերադարձնում պատճեն:

Ստատիկ անդամներ

Դասը կարող է պարունակել ստատիկ անդամներ՝ փոփոխականներ կամ ֆունկցիաներ: Դասի ստատիկ անդամների արժեքը կախված չէ կոնկրետ օբյեկտից և նույնն է ամբողջ դասի համար: Այդ պատճառով ստատիկ փոփոխականներին հաճախ անվանում են **դասի փոփոխականներ**:

Օրինակ՝ գրենք մի ծրագիր, որտեղ դասի ստատիկ անդամը ցույց տա, թե այդ դասային տիպի քանի օբյեկտ է հայտարարված:

```
// klasi statik andamner
#include <iostream>
using namespace std;

class CDummy {
public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
    CDummy a;
    CDummy b[5];
    CDummy * c = new CDummy;
    cout << a.n << endl;
    delete c;
    cout << CDummy::n << endl;
    return 0;
}
```

7
6

Փաստորեն, ստատիկ անդամներն ունեն նույն հատկությունները, ինչ զրոբալ փոփոխականները, բայց գտնվում են դասի տիրույթում: Ընդունված է տալ ստատիկ անդամի նախատիպը (նկարագրությունը) դասի մեջ, իսկ որոշումը (սկզբնարժեքավորումը)՝ դասից դուրս՝ զրոբալ տեսանելիության տիրույթում, ինչպես արված է նախորդ ծրագրում:

Քանի որ ստատիկ անդամը միևնույն փոփոխականն է իր դասի բոլոր օբյեկտների համար, նրան կարելի է դիմել որպես տվյալ դասիային տիպի օբյեկտի անդամ, ինչպես նաև որպես դասի անդամ (սա ճիշտ է **միայն ստատիկ** անդամների համար):

```
cout << a.n;
cout << CDummy::n;
```

Այս երկու կանչերն էլ տպում են միևնույն փոփոխականը՝ **CDummy** դասի **n** ստատիկ փոփոխականը:

Բացի ստատիկ փոփոխականներից դասերը կարող են պարունակել նաև ստատիկ ֆունկցիաներ: Մրանց իմաստը շատ մոտ է ստատիկ փոփոխականների իմաստին՝ սրանք գլոբալ ֆունկցիաներ են, որոնք կանչվում են որպես դասի տվյալ օբյեկտի անդամ-ֆունկցիաներ: Մակայն տրամաբանական է, որ ստատիկ ֆունկցիաները կարող են օգտագործել միայն ստատիկ անդամներ, այսինքն, նրանք չեն կարող դիմել տվյալ դասի ոչ ստատիկ անդամներին, քանի որ դրանք արդեն կախված են օբյեկտից: **this** բանալի-բառը նույնպես չի կարելի օգտագործել ստատիկ ֆունկցիաներում:

```
// klasi statik funkciனர்
#include <iostream>
using namespace std;

class CDummy {
public:
    static int n;
    static void PrintCount();
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;
void CDummy::PrintCount() {
    cout << n << endl;
}

int main () {
    CDummy a;
    CDummy b[5];
    CDummy * c = new CDummy;
    cout << a.n << endl;
    delete c;
    cout << CDummy::n << endl;
    CDummy::PrintCount();
    return 0;
}
```

7
6

Այս օրինակում մենք հայտարարել ենք `PrintCount` անունով ստատիկ ֆունկցիա, որը տպելու է դասի `n` ստատիկ անդամը:

Ցուցիչ դասի անդամների վրա

Դասի անդամների վրա ցուցիչները տարբերվում են այլ ցուցիչներից: Ոչ ստատիկ անդամի կամ ոչ ստատիկ անդամ ֆունկցիայի վրա ցուցիչ հայտարարելու համար հարկավոր է գրել դասի անունը և :: աստղանիշից (*) առաջ: Անդամների վրա ցուցիչները չեն կարող ձևափոխվել սովորական ցուցիչների և հակառակը: Սովորական ցուցիչներ են համարվում նաև դասի ստատիկ անդամների վրա ցույց տվող ցուցիչները:

Դիտարկենք հետևյալ օրինակը.

```
#include <iostream>
using namespace std;

class CPair
{
    int a, b;
public:
    CPair() { a = 0; b = 0; }
    void setA( int val ) { a = val; }
    void setB( int val ) { b = val; }
    int add() const { return a + b; }
    int subtract() const { return a - b; }
};

int getResult( const CPair& p, int (CPair::*fp) () const )
{
    return (p.*fp)();
}

void main()
{
    CPair p;
    p.setA(7);
    p.setB(11);
    cout << "7 + 11 = " << getResult(p,&CPair::add) << endl;
    cout << "7 - 11 = " << getResult(p,&CPair::subtract) << endl;
}
```

```
7 + 11 = 18
7 - 11 = -4
```

Այստեղ մենք հայտարարել ենք CPair անունով դաս, որն ունի երկու ներքին փոփոխականներ և անդամ ֆունկցիաներ՝ այդ փոփոխականներին արժեքներ վերագրելու և դրանց հետ գործողություններ կատարելու համար: Մենք նաև հայտարարել ենք getResult անունով գլոբալ ֆունկցիա, որին որպես պարամետր փոխանցելու ենք CPair տիպի օբյեկտ և CPair դասի գործողության անդամ ֆունկցիայի ցուցիչ: Այդպիսի ցուցիչը, ինչպես արդեն տեսաք, հայտարարվում է հետևյալ կերպ.

```
int (CPair::*fp) () const
```

Ուշադրություն դարձրեք const-ի վրա: Դրա առկայությունը պարտադիր է, քանզի մեր ֆունկցիան const է, այսինքն դրա կանչի արդյունքում օբյեկտի վիճակը չի փոփոխվում:

Վերջապես getResult-ին, անդամ ֆունկցիայի ցուցիչ փոխանցելու համար գրում ենք հետևյալը.

```
& դասի անուն :: անդամի անուն
```

```
ինչպես օրինակ`
```

```
&CPair::add
```

Վերագրման օպերատորի ծանրաբեռնում

Վերագրման օպերատորի (=) նկարագրումը նման է կամայական բինար օպերատորի նկարագրման, հետևյալ բացառություններով`

- Այն պետք է լինի ոչ ստատիկ անդամ ֆունկցիա: Ոչ մի operator= չի կարող նկարագրված լինել որպես դասի ոչ անդամ ֆունկցիա:
- Այն չի ժառանգվում ժառանգող դասի կողմից (Քննարկվելու է միջիջ հետո):
- Եթե այն չի որոշված տվյալ դասի համար, ապա թարգմանիչը կատեղծի դրան ձեր փոխարեն, ինչպես արդեն ասվել է:

```
// vektorner: veragrman operatori orinak
#include <iostream>
using namespace std;

class CVector {
public:
    int x,y;
    CVector () {} ;
    CVector (int,int);
    CVector& operator= (const CVector& param);
};

CVector::CVector (int a, int b) {
    x = a;
    y = b;
}

CVector& CVector::operator= (const CVector& param)
{
    x=param.x;
    y=param.y;
    return *this;
}

int main () {
    CVector a (3,1);
    CVector b = a;
    cout << "(" << b.x << "," << b.y << ")" << endl;
    return 0;
}
```

```
(3,1)
```

Ուշադրություն դարձրեք այն հանգամանքի վրա, որ օպերատորը վերադարձնում է հենց ինքն իրեն: Դա արվում է օպերատորի վարքը պահպանելու համար՝

```
c1 = c2 = c3;
```

որտեղ `c1`, `c2`, `c3` –ը, սվյալ դեպքում, `CVector` տիպի փոփոխականներ են:

Ֆունկցիայի կանչ օպերատորի ծանրաբեռնում

Սա բինար օպերատոր է, որը պետք է հայտարարված լինի որպես դասի ոչ ստատիկ անդամ ֆունկցիա: Դրա ծանրաբեռնումը ոչ մի ազդեցություն չի ունենա այն բանի վրա, թե ինչպես են կանչվելու ֆունկցիաները: Այն ազդելու է միայն սվյալ դասի օբյեկտների նկատմամբ փակագծեր կիրառելիս՝

```
CVector v;  
v( 1,3 );
```

Սովորաբար այսպիսի կոդը անիմաստ է, սակայն մենք կարող ենք օգտագործել այն, օրինակ, մեր վեկտորին ինչ-որ շեղում տալու համար.

```
// vektorner: funkciayi kanchi canrabernman orinak  
#include <iostream>  
using namespace std;  
  
class CVector {  
public:  
    int x,y;  
    CVector () {};  
    CVector (int,int);  
    CVector& operator() (int dx, int dy);  
    void print();  
};  
  
CVector::CVector (int a, int b) {  
    x = a;  
    y = b;  
}  
  
CVector& CVector::operator() (int dx, int dy)  
{  
    x += dx;  
    y += dy;  
    return *this;  
}  
  
void CVector::print()  
{  
    cout << "(" << x << "," << y << ")" << endl;  
}  
  
int main () {  
    CVector v(3,1);  
    v.print();  
    v(5,8);  
}
```

```
v.print();
return 0;
}
(3,1)
(8,9)
```

const բանալի-բառի այլ կիրառությունները

Մինչ այժմ մենք տեսանք այս բանալի բառի կիրառությունը հաստատուն փոփոխականներ, հաստատուն փոփոխականների վրա ցուցիչներ և հաստատուն հասցեի վրա ցույց տվող ցուցիչներ հայտարարելու համար: Սակայն այն սրանով չի սահմանափակվում:

const կարող են լինել ֆունկցիայի արգումենտները, զգուշացնելու համար, որ ֆունկցիայի կանչի արդյունքում արգումենտը չի փոփոխվելու: Դիտարկենք հետևյալ ֆունկցիաների նախատիպերը.

```
void f1(const int& x);
```

x-ը չի փոփոխվելու ֆունկցիայի ներսում:

```
void f2(const int x);
```

ավելորդ է, քանի որ x-ը արդեն իսկ փոխանցվում է արժեքով:

```
void f3(int* const y);
```

ավելորդ է, քանի որ y-ի հասցեն արդեն իսկ փոխանցվում է արժեքով (հիշեք, որ տվյալ դեպքում y-ը հաստատուն հասցեի վրա ցույց տվող ցուցիչ է):

```
void f4(int* const *y);
```

ցուցիչը փոխանցվում է ցուցիչով, ընդ որում y-ի ցույց տված հասցեի պարունակությունը *y-ը, չի կարող փոփոխվել:

```
void f4(const int*& y);
```

Իմաստ ունի գրել. ցուցիչը փոխանցվում է հասցեով:

Նկատի ունեցեք նաև, որ որպես պարամետր ամպայման չէ փոխանցել const փոփոխականներ, այսինքն, մենք կարող ենք գրել հետևյալը.

```
int a = 5;
f1(a);
```

Ֆունկցիայի արգումենտները const դարձնելով, սակայն, մենք հանգում ենք մեկ այլ խոչընդոտի. առաջանում է պրոբլեմ const պարամետրի վրա ցուցիչի կամ հղման վերադարձման հետ: Դիտարկենք հետևյալ օրինակը.

```
int* f(const int* x)
{
    return x;
}
```

Սա անթույլատրելի է, քանզի, հակառակ դեպքում, հնարավոր կդառնա փոփոխել, որպես պարամետր փոխանցված փոփոխականի արժեքը, մինչդեռ ֆունկցիան խոստանում էր, որ իր կանչի արդյունքում փոփոխականը կմնա անփոփոխ: Ձեր մոտ կարող է առաջանա մի հարց, բայց ինչ էթե փոփոխականը հաստատուն չէ, ինչու այն չի կարող վերադարձվել ֆունկցիայի կողմից և լինել փոփոխման ենթակա, չէ որ ֆունկցիայի ներսում փոփոխականը չի փոփոխվում: Պատճառն այն է, որ հենց դրանում է կայանում C++ -ի էությունը՝ յուրաքանչյուր տիպի պետք է համապատասխանի ճշգրիտ իր տիպը, կամ պետք է գոյություն ունենա համապատասխան ձևափոխման օպերատոր: Մեր դեպքում `int*`-ը և `const int*`-ը տարբեր տիպեր են, և `const int*`-ից `int*` սահմանած ձևափոխման օպերատոր չկա:

C++ -ում կա այս հարցի լուծման երեք եղանակ: Առաջին եղանակը՝ `const`-ի վերացումն է ֆունկցիայի նկարագրությունից: Թեև, թվում է, որ սա հարցի լուծման ամենահեշտ եղանակն է, սակայն այն հակասում է ընդունված ստանդարտներին: Եթե ֆունկցիան չի փոփոխում իր պարամետրը, ապա վերջինս պետք է լինի `const`:

Այս հարցի լուծման երկրորդ եղանակը՝ `const_cast` օպերատորի օգտագործումն է, որը թույլ է տալիս ազատել փոփոխականին իր `const` հատկությունից, եթե այն առկա է: Դրա օգտագործման եղանակն հետևյալն է՝

```
const_cast< տիպ >( փոփոխական )
```

որտեղ `տիպ`՝ փոխակերպվող տիպն է, իսկ `փոփոխականը`՝ հաստատուն փոփոխականի անունը: Վերևի օրինակի ճիշտ լուծումը այս եղանակով կլինի.

```
int* f(const int* x)
{
    return const_cast<int*>(x);
}
```

Սա ավելի բարդ լուծում է, սակայն այն նույնպես չի կարող համարվել լավագույն լուծումը, քանզի սա ավելի շատ հարկադրական եղանակ է:

Վերջապես, երրորդ՝ ամենալավ լուծումը, վերադարձվող արժեքի հաստատումացումն է: Դա թույլ կտա առանց որևէ բարդության վերադարձնել `const` փոփոխականներ և պաշտպանել դրանց հետագա փոփոխություններից.

```
const int* f(const int* x)
{
    return x;
}
```

Այսպիսով մենք տեսանք `const`-ի կիրառությունները ֆունկցիաներում: Սակայն սա դեռ ամենը չէ `const`-ի մասին: Դիտարկենք հետևյալ դասը.

```

class Point
{
    int x, y;
public:
    Point(int a, int b)
    {
        x = a; y = b;
    }
    void setPoint(int a, int b)
    {
        x = a; y = b;
    }
    void showPoint()
    {
        cout << x << ", " << y << endl;
    }
    const Point* getClosestPoint(const Point &p)
    {
        return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p;
    }
};

```

Սա երկչափ կետի դաս է, որը բաղկացած է կառուցիչից և վերագրող, տպող ու (0,0) կետին ամենամոտիկ կետը վերադարձնող ֆունկցիաներից: Այժմ հայտարարենք այդ դասի օբյեկտ՝ p1, հաստատուն ցուցիչ՝ p2, և կանչենք p1-ի getClosestPoint ֆունկցիան, որպես պարամետր տալով (3,3) կետը՝ Point(3,3).

```

Point p1(1,2);
const Point* p2;
p2 = p1.getClosestPoint(Point(3,3));

```

getClosestPoint անդամ ֆունկցիան կվերադարձնի դրանցից (0,0)-ին ամենամոտիկ կետի վրա ցույց տվող ցուցիչ, որը պաշտպանված կլինի որևէ փոփոխություններից: Դա նշանակում է, որ մենք չենք կարողանա դիմել այդ օբյեկտի setPoint անդամ ֆունկցիային՝

```
p2->setPoint(1,1);
```

քանի որ դրա վիճակը հնարավոր է փոփոխվի ֆունկցիայի կանչի արդյունքում, մինչդեռ այն const է: Սակայն նույնը կկատարվի նաև showPoint ֆունկցիային դիմելիս՝

```
p2->showPoint();
```

Հնայած, որ այն հաստատ չի փոփոխելու տրված օբյեկտի վիճակը:

Բարեբախտաբար, C++ -ը մեզ այս հարցի լուծման համար ևս նախատեսել է const բանալի բառը: Ֆունկցիայի նկարագրման մեջ այն տեղադրելով պարամետրերի ցուցակի փակվող փակագծի և ; սիմվոլի միջև, իսկ ֆունկցիայի հայտարարման մեջ՝ փակվող փակագծի և բացվող ձևավոր փակագծի միջև, դուք հասկացնել կտաք թարգմանիչին, որ տրված անդամ ֆունկցիան չի փոփոխի տվյալ օբյեկտի վիճակը.

```

class Point
{

```

```

        int x, y;
public:
    Point(int a, int b)
    {
        x = a; y = b;
    }
    void setPoint(int a, int b)
    {
        x = a; y = b;
    }
    void showPoint() const
    {
        cout << x << ", " << y << endl;
    }
    const Point* getClosestPoint(const Point &p)
    {
        return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p;
    }
};

```

Որից հետո էլ, հանգիստ կկարողանաք դիմել showPoint ֆունկցիային՝

```

Point p1(1,2);
const Point* p2;
p2 = p1.getClosestPoint(Point(3,3));
p2->showPoint();

```

Դասերի օգտագործման օրինակ՝ ռացիոնալ թվերի դաս

Այժմ գրենք ռացիոնալ թվերի դաս, որի միջոցով մենք կկարողանանք հայտարարել ռացիոնալ թվեր և կատարել գործողություններ դրանց հետ.

```

// Rational class
#include <iostream>
using namespace std;

class Rational {
    long nmr, dnm;
    void normalize();

public:
    Rational(long n=0, long d=1)
    {
        nmr = n; dnm = d;
        this->normalize();
    }

    Rational operator + (const Rational &x) const;
    Rational operator - (const Rational &x) const;
    Rational operator * (const Rational &x) const;
    Rational operator / (const Rational &x) const;

    void operator += (const Rational &x);
    void operator -= (const Rational &x);
    void operator *= (const Rational &x);
    void operator /= (const Rational &x);

```

```

    bool operator == (const Rational &other) const;
    bool operator < (const Rational &other) const;
    bool operator > (const Rational &other) const;
    void show() const;
};

Rational Rational::operator + (const Rational &x) const
{
    return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm);
}
Rational Rational::operator - (const Rational &x) const
{
    return Rational(nmr*x.dnm - x.nmr*dnm, dnm*x.dnm);
}

Rational Rational::operator * (const Rational &x) const
{
    return Rational(nmr * x.nmr, dnm * x.dnm);
}

Rational Rational::operator / (const Rational &x) const
{
    return Rational(nmr * x.dnm, dnm * x.nmr);
}

void Rational::operator += (const Rational &x)
{
    nmr = nmr * x.dnm + x.nmr * dnm;
    dnm = dnm * x.dnm;
    this->normalize();
}

void Rational::operator -= (const Rational &x)
{
    nmr = nmr * x.dnm - x.nmr * dnm;
    dnm = dnm * x.dnm;
    this->normalize();
}

void Rational::operator *= (const Rational &x)
{
    nmr = nmr * x.nmr;    dnm = dnm * x.dnm;
    this->normalize();
}

void Rational::operator /= (const Rational &x)
{
    nmr = nmr * x.dnm;    dnm = dnm * x.nmr;
    this->normalize();
}

bool Rational::operator == (const Rational &other) const
{
    return (nmr * other.dnm == dnm * other.nmr);
}

bool Rational::operator < (const Rational &other) const
{
    return (nmr * other.dnm < dnm * other.nmr);
}

```



```

bool Rational::operator > (const Rational &other) const
{
    return (nmr * other.dnm > dnm * other.nmr);
}

void Rational::show() const
{
    cout << " " << nmr << "/" << dnm;
}

void Rational::normalize()
{
    if (nmr == 0) {
        dnm = 1; return;
    }
    int sign = 1;
    if (nmr < 0) {
        sign = -1; nmr = -nmr;
    }
    if (dnm < 0) {
        sign = -sign; dnm = -dnm;
    }
    long gcd = nmr, value = dnm;
    while (value != gcd) {
        if (gcd > value)
            gcd = gcd - value;
        else
            value = value - gcd;
    }
    nmr = sign * (nmr/gcd); dnm = dnm/gcd;
}

int main()
{
    Rational a(1,4), b(3,2), c, d;
    c = a + b;
    a.show(); cout << " +"; b.show(); cout << " ="; c.show();
    cout << endl;
    d = b - 7;
    b.show(); cout << " - " << 7 << " ="; d.show(); cout << endl;
    c = a * 3;
    a.show(); cout << " * " << 3 << " ="; c.show(); cout << endl;
    d = b / 2;
    b.show(); cout << " / " << 2 << " ="; d.show(); cout << endl;
    c.show();
    c += 3;
    cout << " += " << 3 << " ="; c.show(); cout << endl;
    d.show();
    d *= 2;
    cout << " *= " << 2 << " ="; d.show(); cout << endl;
    if (b < 2) {
        b.show();
        cout << " < " << 2 << endl;
    }
    return 0;
}

```

```

1/4 + 3/2 = 7/4
3/2 - 7 = -11/2

```

$1/4 * 3 = 3/4$
$3/2 / 2 = 3/4$
$3/4 += 3 = 15/4$
$3/4 *= 2 = 3/2$
$3/2 < 2$

Այս օրինակում հայտարարված է Rational անունով ռացիոնալ թվերի դաս, որը ունի երկու ներքին փոփոխականներ՝ nmr (nominator՝ համարիչ) և dnm (denominator՝ հայտարար), normalize անունով ներքին ֆունկցիա, որը պատասխանատու կլինի ռացիոնալ թվի վերջնական տեսքի բերման, show գլոբալ ֆունկցիա, որի միջոցով ռացիոնալ թիվը տպվելու է էկրանին և բազմաթիվ օպերատորներ +, -, *, /, +=, -=, *=, /=, ==, <, >:

Եկեք մանրակրկիտ զննենք այս օրինակը: նախ, ուշադրություն դարձրեք, որ դասի որոշ անդամ ֆունկցիաները հայտարարված են const, քանի որ դրանք չեն փոփոխում տրված դասի օբյեկտի վիճակը: const են հայտարարված նաև բոլոր պարամետրները, որոնք չեն փոփոխվում ֆունկցիայի կանչի արդյունքում: Առանձնահատուկ ուշադրություն է պահանջում նաև կառուցիչը.

```
Rational(long n=0, long d=1)
{
    nmr = n; dnm = d;
    this->normalize();
}
```

Այն պարունակում է լրուցյալ արգումենտներ: Իսկ ի՞նչ կկատարվի, եթե մենք այն փոխարինանք երկու կառուցիչներով՝ դասարկ կառուցիչով, և նույն կառուցիչով, առանց լրուցյալ արգումենտների.

```
Rational()
{
    nmr = 0; dnm = 1;
    this->normalize();
}
```

```
Rational(long n, long d)
{
    nmr = n; dnm = d;
    this->normalize();
}
```

Թվում է, թե ոչ մի բան չի փոփոխվի, բայց ոչ: Թարգմանիչը կտա սխալ հետևյալ տողի վրա.

```
d = b - 7;
```

Ինչու՞: Չէ՞, որ մեր հանում գործողությունը ոչ մի կապ չունի կառուցիչի հետ, և ավելին, ինչպե՞ս էր դա կատարվում մինչ այս փոփոխությունները, եթե մենք սահմանել ենք միայն հանում գործողություն դասի օբյեկտների միջև, այլ ոչ դասի օբյեկտի և ամբողջ թվի միջև: Իրականում, երբ թարգմանիչը մշակում է `b - 7` արտահայտությունը, այն, սկզբում, փորձում է գտնել մի ֆունկցիա, որը պատասխանատու է լինելու կատարելու հանում գործողությունը տրված տիպերի

համար: Այդպիսի ֆունկցիա չգտնելու դեպքում, այն դիտարկում է b օբյեկտի տիպի համար որոշված հանում գործողությունները և փորձում ձևափոխել երկրորդ պարամետրը՝ 7 թիվը, այդ գտնված ֆունկցիայի երկրորդ արգումենտի տիպին, մեր դեպքում՝ Rational տիպին: Այսինքն թարգմանիչի համար $b - 7$ արտահայտությունը կվերածվի հետևյալին.

```
b.operator -( (Rational) 7 );
```

Այժմ, վերհիշելով *տիպերի ձևափոխման օպերատորները* (բաժին 1.3), վերը նշված արտահայտությունը կարող ենք ներկայացնել այսպես.

```
b.operator -( Rational(7) );
```

Փաստորեն մենք ստացանք կանչ կառուցիչին: Իսկ, քանի որ, մեր սկզբնական կառուցիչի ձևափոխումից հետո մենք այլևս չունենք այսպիսի պայմաններին բավարարող կառուցիչ, թարգմանիչը չի կարողանա ձևափոխել int տիպը Rational տիպին:

Այսպիսով, մեր բերված ռացիոնալ թվերի դասի կառուցիչը բացի նրանից, որ հիմնական ու լրությամբ կառուցիչ է, այն նաև ծառայում է, որպես ձևափոխման կառուցիչ:

Սակայն, այժմ առաջանում է մեկ այլ հարց: Մենք հասկացանք, որ $b - 7$ արտահայտությունը վերածվում է b օբյեկտի հանում օպերատոր անդամ ֆունկցիայի կանչին, բայց ի՞նչ, եթե գրված է $7 - b$: Չէ՞ որ այդ դեպքում արդեն չի կանչվի b -ի անդամ ֆունկցիան: Այս հարցի լուծումը կտրվի հաջորդ բաժնում:

Բաժին 4.3

Դասերի հարաբերությունները

Բարեկամ ֆունկցիաներ (Friend Functions)

4.1 բաժնում տեսանք, որ գոյություն ունեն դասի անդամների պաշտպանության երեք աստիճաններ՝ **public**, **protected** և **private**: Այն դեպքում, երբ դասի անդամը հայտարարված է որպես **protected** կամ **private**, նրան չի կարելի դիմել տվյալ դասի սահմաններից դուրս: Մակայն երբեմն այս կանոնը խախտելու անհրաժեշտություն է առաջանում: Դա կատարելու համար օգտագործում ենք **friend** բանալի-բառը: Որպեսզի դրսի ֆունկցիան կարողանա օգտագործել դասի **private** և **protected** անդամները, պետք է գրենք այդ ֆունկցիայի նախաստիպը այն դասի մեջ, որի **private** և **protected** անդամները մենք ուզում ենք օգտագործել՝ գրելով **friend** բառը նախաստիպի սկզբից: Դրանով ֆունկցիան հայտարարվում է դասի բարեկամ: Հետևյալ օրինակում հայտարարում ենք **duplicate** բարեկամ ֆունկցիա:

```
// barekam funkcianer
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
    width = a;
    height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
    CRectangle rectres;
    rectres.width = rectparam.width*2;
    rectres.height = rectparam.height*2;
    return (rectres);
}

int main () {
    CRectangle rect, rectb;
    rect.set_values (2,3);
    rectb = duplicate (rect);
    cout << rectb.area();
}
```

24

duplicate ֆունկցիայից կարող ենք դիմել **CRectangle** դասի **width** և **height** փոփոխականներին, որոնք գտնվում են **private** տեսանելիության տիրույթում: Ուշադրություն դարձրեք նաև այն բանին, որ **duplicate()** ֆունկցիան կանչում ենք ինչպես սովորական գործալ ֆունկցիա, այլ ոչ որպես **CRectangle** դասի անդամ:

Բարեկամ ֆունկցիաները կարող են օգտակար լինել, օրինակ, երկու տարբեր դասերի միջև գործողություններ իրականացնելիս: Բարեկամ ֆունկցիաների օգտագործումը համարվում է օբյեկտակողմնորոշված ծրագրավորման մոտեցումներից դուրս, այդ պատճառով, խորհուրդ է տրվում հնարավորության դեպքում խուսափել դրանց օգտագործումից: Օրինակ՝ նախորդ ծրագրում ավելի ճիշտ կլիներ իրականացնել `duplicate()` ֆունկցիան `CRectangle` դասի մեջ:

Ռացիոնալ թվերի դաս

Բարեկամ ֆունկցիաների միջոցով մենք կարող ենք բերել մեր ռացիոնալ թվերի դասը վերջնական տեսքի, այս անգամ արդեն, թույլ տալով, ինչպես ռացիոնալ թվերին գումարել այլ թվեր, այնպես էլ այլ թվերին գումարել ռացիոնալ թվեր:

```
#include <iostream>
using namespace std;

class Rational {
    long nmr, dnm;
    void normalize();
public:
    Rational(long n=0, long d=1)
    {
        nmr = n;  dnm = d;
        this->normalize();
    }

    friend Rational operator +
        (const Rational &x, const Rational &y);
    friend Rational operator -
        (const Rational &x, const Rational &y);
    friend Rational operator *
        (const Rational &x, const Rational &y);
    friend Rational operator /
        (const Rational &x, const Rational &y);

    friend void operator += (Rational &x, const Rational &y);
    friend void operator -= (Rational &x, const Rational &y);
    friend void operator *= (Rational &x, const Rational &y);
    friend void operator /= (Rational &x, const Rational &y);

    friend bool operator == (const Rational &x, const Rational &y);
    friend bool operator <  (const Rational &x, const Rational &y);
    friend bool operator >  (const Rational &x, const Rational &y);
    void show() const;
};

Rational operator + (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm + y.nmr * x.dnm, x.dnm * y.dnm);
}
Rational operator - (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm - y.nmr * x.dnm, x.dnm * y.dnm);
}

Rational operator * (const Rational &x, const Rational &y)
```

```

{
    return Rational(x.nmr * y.nmr, x.dnm * y.dnm);
}

Rational operator / (const Rational &x, const Rational &y)
{
    return Rational(x.nmr * y.dnm, x.dnm * y.nmr);
}

void operator += (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.dnm + y.nmr * x.dnm;
    x.dnm = x.dnm * y.dnm;
    x.normalize();
}

void operator -= (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.dnm - y.nmr * x.dnm;
    x.dnm = x.dnm * y.dnm;
    x.normalize();
}

void operator *= (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.nmr;  x.dnm = x.dnm * y.dnm;
    x.normalize();
}

void operator /= (Rational &x, const Rational &y)
{
    x.nmr = x.nmr * y.dnm;  x.dnm = x.dnm * y.nmr;
    x.normalize();
}

bool operator == (const Rational &x, const Rational &y)
{
    return (x.nmr * y.dnm == x.dnm * y.nmr);
}

bool operator < (const Rational &x, const Rational &y)
{
    return (x.nmr * y.dnm < x.dnm * y.nmr);
}

bool operator > (const Rational &x, const Rational &y)
{
    return (x.nmr * y.dnm > x.dnm * y.nmr);
}

void Rational::show() const
{
    cout << " " << nmr << "/" << dnm;
}

void Rational::normalize()
{
    if (nmr == 0) {
        dnm = 1;  return;
    }
}

```

```

int sign = 1;
if (nmr < 0) {
    sign = -1; nmr = -nmr;
}
if (dnm < 0) {
    sign = -sign; dnm = -dnm;
}
long gcd = nmr, value = dnm;
while (value != gcd) {
    if (gcd > value)
        gcd = gcd - value;
    else
        value = value - gcd;
}
nmr = sign * (nmr/gcd); dnm = dnm/gcd;
}

int main()
{
    Rational a(1,4), b(3,2), c, d;
    c = 5 + a;
    cout << " " << 5 << " +"; a.show(); cout << " =";
    c.show(); cout << endl;
    d = 1 - b; // operator-(Rational(1),b);
    cout << " 1 -"; b.show(); cout << " ="; d.show(); cout << endl;
    c = 7 * a; // operator*(Rational(7),a);
    cout << " 7 *"; a.show(); cout << " ="; c.show(); cout << endl;
    d = 2 / b; // operator/(Rational(2),b);
    cout << " 2 /"; b.show(); cout << " ="; d.show(); cout << endl;
    c.show();
    c += 3; // operator+=(c,Rational(3));
    cout << " += " << 3 << " ="; c.show(); cout << endl;
    d.show();
    d *= 2; // operator*=(d,Rational(2))
    cout << " *= " << 2 << " ="; d.show(); cout << endl;
    if (a < 5) cout << " a < 5\n"; // operator<(a,Rational(5));
    if (1 < b) cout << " 1 < b\n"; // operator<(Rational(1),b);
    if (1 < 5) cout << " 1 < 5\n"; // built-in inequality operator
    if (d * b - a == c - 1) cout << " d*b-a == c-1 ==";
    (c - 1).show(); cout << endl;
    return 0;
}

```

```

5 + 1/4 = 21/4
1 - 3/2 = -1/2
7 * 1/4 = 7/4
2 / 3/2 = 4/3
7/4 += 3 = 19/4
4/3 *= 2 = 8/3
a < 5
1 < b
1 < 5
d*b-a == c-1 == 15/4

```

Բարեկամ դասեր (Friend Classes)

Բացի բարեկամ ֆունկցիաներից C++ - ում իրականացված է նաև բարեկամ դասերի գաղափարը: Մի դասը հայտարարելով մյուսի բարեկամ առաջինի անդամ-

Ֆունկցիաներին հնարավորություն ենք տալիս դիմելու երկրորդ դասի **private** և **protected** անդամներին.

```
// barekam klas
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
public:
    int area (void)
        {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
private:
    int side;
public:
    void set_side (int a)
        {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
    width = a.side;
    height = a.side;
}

int main () {
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

16

Այս օրինակում հայտարարեցինք **CRectangle** - ը որպես **CSquare** դասի բարեկամ՝ թույլ տալով **CRectangle** - ին դիմել **CSquare** դասի **private** և **protected** անդամներին, իսկ ավելի կոնկրետ՝ **CSquare::side** - ին, որը ցույց է տալիս քառակուսու կողի երկարությունը:

Այս ծրագրում կա մի նորություն ևս՝

```
class CSquare;
```

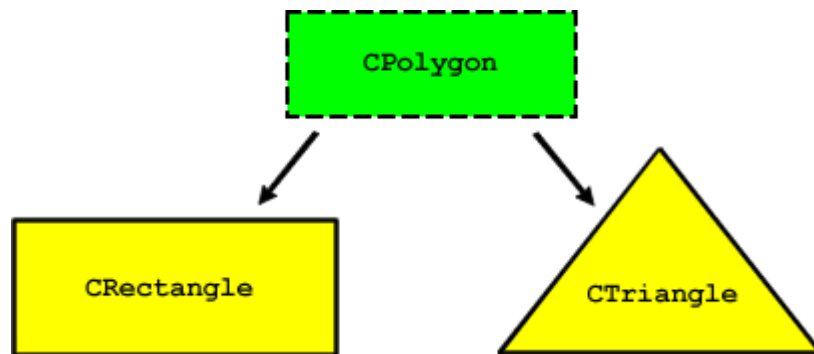
Սա **CSquare** դասի նախատիպն է: Այս ծրագրում այն պարտադիր է, քանի որ **CRectangle** դասում օգտագործում ենք **CSquare** - ը, իսկ **CSquare** – ում՝ **CRectangle** – ը:

Ուշադրություն դարձնենք այն փաստին, որ այս օրինակում **CRectangle** – ն է հայտարարված **CSquare** դասի բարեկամ և ոչ հակառակը: Որպեսզի **CSquare** – ից կարողանանք դիմել **CRectangle** – ի **private** և **protected** անդամներին, պիտի **CSquare** - ը հայտարարենք **CRectangle** – ի բարեկամ:

Դասերի ժառանգականությունը (Inheritance Between Classes)

Դասերի կարևորագույն հատկություններից է ժառանգականությունը: Այն թույլ է տալիս ստեղծել մի օբյեկտ, որը հիմնված է մեկ այլ օբյեկտի վրա: Այս դեպքում ասում ենք, որ առաջին դասը ժառանգված է երկրորդից: Ստացված դասը պարունակում է սկզբնականի բոլոր անդամները, ինչպես նաև իր սեփականները: Դիցուք ուզում ենք հայտարարել ուղղանկյուն **CRectangle** և եռանկյուն **CTriangle** դասերը: Դրանք ունեն ընդհանուր հատկություններ: Օրինակ՝ երկուսն էլ ունեն հիմք և բարձրություն, մակերես, ... :

Դասերի աշխարհում **CRectangle** և **CTriangle** դասերի ընդհանուր մասը կարողք ենք նկարագրել ընդհանուր **CPolygon** դասով:



CPolygon դասը կպարունակի միայն այն անդամները, որոնք ընդհանուր են բոլոր բազմանկյունների համար, մեր դեպքում՝ բարձրությունը (**height**) և լայնությունը (**width**):

Որպեսզի մի դասը ժառանգի մեկ այլ դասի, պետք է օգտագործենք : օպերատորը ժառանգող դասի հայտարարության մեջ հետևյալ կերպ.

```
class դասի անուն : public ժառանգվող դասի անուն;
```

Այստեղ **public** - ը կարող է փոխարինվել հետևյալ տեսանելիության նշիչներով՝ **protected** կամ **private**:

Օրինակ.

```
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
```

```

    { width=a; height=b;}
};

class CRectangle : public CPolygon {
public:
    int area (void)
    { return (width * height); }
};

class CTriangle : public CPolygon {
public:
    int area (void)
    { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}

```

20
10

Ինչպես երևում է այս օրինակից, **CRectangle** և **CTriangle** դասերի օբյեկտներից յուրաքանչյուրը պարունայում է **CPolygon** դասի անդամները՝ **width**, **height** և **set_values()**:

Նշենք, որ ժառանգող դասի անդամները չեն կարող դիմել ժառանգվող դասի **private** անդամներին: Հենց սրանով է **protected** նշիչը տարբերվում **private** – ից: Քանի որ օրինակում ցանկանում էինք **CPolygon** դասի **width** և **height** անդամները ձևափոխել **CRectangle** և **CTriangle** դասերի անդամ-ֆունկցիաներում, **CPolygon** դասի անդամները հայտարարեցինք **protected**:

Ստորև ամփոփված է, թե ովքեր կարող են դիմել ժառանգվող դասի անդամներին, դրանց՝ տարբեր տեսանելիության տիրույթներում գտնվելու դեպքերում:

Դիմում	public	protected	private
Նույն դասի անդամները	այո	այո	այո
Ժառանգող դասի անդամները	այո	այո	ոչ
Ոչ անդամները	այո	ոչ	ոչ

Այստեղ «ոչ անդամներ» են համարվում տվյալ դասերից դուրս գտնվող բոլոր ֆունկցիաները, ինչպիսին է, օրինակ, **main()**-ը:

Մեր օրինակում **CRectangle** և **CTriangle** դասերի կողմից ժառանգված անդամները չեն փոխել իրենց տեսանելիության տիրույթը, այսինքն՝

```
CPolygon::width           // protected e
CRectangle::width        // protected e

CPolygon::set_values()    // public e
CRectangle::set_values() // public e
```

Դա այդպես է, քանի որ **CPolygon** կլասը ժառանգել ենք որպես **public**՝

```
class CRectangle: public CPolygon;
```

Հետևյալ ցուցակը ցույց է տալիս, թե ինչպիսին են լինելու ժառանգվող դասի անդամների տեսանելիության տիրույթները ժառանգող դասում՝ անդամների և ժառանգության տիպի տարբեր դեպքերում:

Ժառանգության տեսակ	Երբ ժառանգվող դասի անդամը private է	Երբ ժառանգվող դասի անդամը protected է	Երբ ժառանգվող դասի անդամը public է
private	անհասանելի է	private	private
protected	անհասանելի է	protected	protected
public	անհասանելի է	protected	public

Ի՞նչ է ժառանգվում ժառանգվող դասից

Ժառանգվող կլասի գրեթե յուրաքանչյուր անդամ ժառանգվում է ժառանգող կլասի կողմից, բացի

- կառուցիչից և փլուզիչից
- **operator=()** անդամից
- բարեկամներից

Չնայած նրան, որ ժառանգվող դասի կառուցիչը և փլուզիչը չեն ժառանգվում, սակայն ժառանգվող դասի լրության կառուցիչը (առանց պարամետրերի կառուցիչը) և փլուզիչը միշտ կանչվում են ժառանգող դասի նոր օբյեկտներ ստեղծելիս կամ ջնջելիս:

Եթե ժառանգվող դասը չունի լրության կառուցիչ, կամ եթե ուզում ենք, որ կանչվի գերբեռնված կառուցիչ, երբ ստեղծվում է ժառանգող դասի նոր օբյեկտ, կարող ենք սահմանել այն ժառանգող դասի յուրաքանչյուր կառուցիչի հայտարարության ժամանակ հետևյալ կերպ.

```
ժառանգող կլասի անուն (պարամետրեր) : ժառանգվող կլասի անուն (պարամետրեր) {}
```

Օրինակ.

```

#include <iostream>
using namespace std;

class mother {
public:
    mother ()
        { cout << "mother: aranc parametreri\n"; }
    mother (int a)
        { cout << "mother: int parametrov\n"; }
};

class daughter : public mother {
public:
    daughter (int a)
        { cout << "daughter: int parametrov\n\n"; }
};

class son : public mother {
public:
    son (int a) : mother (a)
        { cout << "son: int parametrov\n\n"; }
};

int main () {
    daughter cynthia (1);
    son daniel(1);

    return 0;
}

```

```

mother: aranc parametreri
daughter: int parametrov

mother: int parametrov
son: int parametrov

```

Այս օրինակում **daughter** դասի օբյեկտ հայտարարելիսուց կանչվում է **mother** դասի դատարկ կառուցիչը, իսկ **son** դասի օբյեկտ հայտարարելիս՝ **mother** դասի գերբեռնված կառուցիչը: Մա տեղի է ունենում **daughter** և **son** դասերի կառուցիչների հայտարարության տարբերության պատճառով.

```

daughter (int a)
son (int a) : mother (a)

```

Նշենք նաև, որ չենք կարող չգերբեռնել **son** դասի կառուցիչը, եթե ուզում ենք, որ օգտագործվի **mother** դասի գերբեռնված կառուցիչը: Այսինքն՝ չենք կարող գրել

```

son () : mother (a)

```

քանզի այստեղ **a** - ն չորոշված փոփոխական է: Մակայն կարող ենք գերբեռնել **son** դասի կառուցիչը 2 պարամետրերով, բայց կանչել **mother** դասի մեկ պարամետրանոց կառուցիչը: Օրինակ՝

```

son (int a, int b) : mother (a)

```

Բազմակի ժառանգականություն (Multiple Inheritance)

C++ - ում հնարավոր է, որ դասը ժառանգի մի քանի դասեր: Դա կատարելու համար դասի հայտարարության ժամանակ պետք է առանձնացնել ժառանգվող դասերի անունները ստորակետներով: Օրինակ՝ եթե ունենք մի դաս (**COutput**), որն օգտագործվում է էկրանին ինչ-որ բան տպելու համար, և ուզում ենք, որ **CRectangle** և **CTriangle** դասերը ժառանգեն այն **CPolygon** դասի հետ մեկտեղ, պետք է գրենք.

```
class CRectangle: public CPolygon, public COutput { ... };  
class CTriangle: public CPolygon, public COutput { ... };
```

Ստորև բերված է լրիվ օրինակը.

```
#include <iostream>  
using namespace std;  
  
class CPolygon {  
protected:  
    int width, height;  
public:  
    void set_values (int a, int b)  
        { width=a; height=b; }  
};  
  
class COutput {  
public:  
    void output (int i);  
};  
  
void COutput::output (int i) {  
    cout << i << endl;  
}  
  
class CRectangle: public CPolygon, public COutput {  
public:  
    int area (void)  
        { return (width * height); }  
};  
  
class CTriangle: public CPolygon, public COutput {  
public:  
    int area (void)  
        { return (width * height / 2); }  
};  
  
int main () {  
    CRectangle rect;  
    CTriangle trgl;  
    rect.set_values (4,5);  
    trgl.set_values (4,5);  
    rect.output (rect.area());  
    trgl.output (trgl.area());  
    return 0;  
}
```

20

10

Բաժին 4.4

Բազմաձևություն (Polymorphism)

Ժառանգվող դասերի ցուցիչներ

Դասերի ժառանգականության ամենակարևոր առավելություններից մեկն այն է, որ ժառանգող դասի ցուցիչի տիպը համատեղելի է ժառանգվող դասի ցուցիչի տիպի հետ: Ստորև բերված է նախորդ բաժիններում քննարկված ուղղանկյան և եռանկյան ծրագիրը, որն օգտագործում է այս հատկությունը:

```
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
};

class CRectangle: public CPolygon {
public:
    int area (void)
        { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
        { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

20
10

main ֆունկցիայում հայտարարվում են **CPolygon** տիպի երկու ցուցիչներ՝ ***ppoly1** և ***ppoly2**: Հետո դրանց համապատասխանաբար վերագրվում են **rect** - ի և **trgl** - ի հասցեները, և քանի որ դրանք **CPolygon** դասից ծնված օբյեկտներ են, այս վերագրումը թույլատրելի է:

Միակ սահմանափակումը, որն առկա է **rect** - ի և **trgl** - ի փոխարեն ***ppoly1** - ի և ***ppoly2** - ի օգտագործման մեջ, այն է, որ և ***ppoly1** - ն, և ***ppoly2** - ը **CPolygon*** տիպի են, և հետևաբար դրանց միջոցով կարող ենք օգտագործել **CRectangle** և **CTriangle**

դասերի միայն այն անդամները, որոնք ժառանգվում են **CPolygon** դասից: Այդ է պատճառը, որ **area()** ֆունկցիան կանչելիս չօգտագործեցինք ***ppoly1** և ***ppoly2** օբյեկտները: Որպեսզի կարողանանք **CPolygon** դասի ցուցիչով կանչել **area()** ֆունկցիան, անհրաժեշտ է, որ այն հայտարարված լինի **CPolygon** դասում:

Վիրտուալ անդամներ

Վիրտուալ անդամներն այն անդամ ֆունկցիաներն են, որոնք կարող են վերահայտարարվել ժառանգող դասում: Այն դեպքում, երբ ժառանգվող դասի ցուցիչով դիմենք ժառանգող դասի օբյեկտի անդամին, որը ժառանգվող դասում հայտարարված էր **virtual**, ապա ժառանգող դասում այդ ֆունկցիայի առկայության դեպքում կկանչվի այդ ֆունկցիան:

virtual անդամ հայտարարելու համար անհրաժեշտ է ժառանգվող դասում՝ անդամի հայտարարության սկզբում, ավելացնել **virtual** բանալի-բառը:

Դիտարկենք հետևյալ օրինակը.

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void)
        { return (0); }
};

class CRectangle: public CPolygon {
public:
    int area (void)
        { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
        { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
```

```

cout << ppoly3->area() << endl;
return 0;
}
20
10
0

```

Այս դեպքում բոլոր երեք դասերը (**CPolygon**, **CRectangle** և **CTriangle**) ունեն միևնույն անդամները՝ **width**, **height**, **set_values()** և **area()**:

area() անդամը հայտարարված է **virtual**, որպեսզի այն կարողանանք վերահայտարարել ժառանգող դասերում: Կարող ենք ստուգել, որ եթե ջնջենք **virtual** բառը և աշխատեցնենք ծրագիրը, ապա արդյունքը բոլոր երեք դեպքերում կլինի **0` 20, 10, 0** - ի փոխարեն: Դա տեղի է ունենում այն պատճառով, որովհետև երեք դեպքերում, համապատասխանաբար **CRectangle::area()**, **CTriangle::area()** և **CPolygon::area()** ֆունկցիաները կանչելու փոխարեն կկանչվի միայն **CPolygon::area()** ֆունկցիան, որն էլ և կվերադարձնի **0**:

Այսպիսով **virtual** - ի իմաստն այն է, որ այն թողնում է ժառանգող դասում հայտարարել անդամ, որի անունով անդամ արդեն հայտարարված է ժառանգվող դասում, և երբ այդ անդամը կանչվում է ժառանգվող դասի ցուցի միջոցով, ժառանգվող դասի անդամի փոխարեն կանչվում է ժառանգող դասում վերահայտարարված անդամը (տե՛ս նախորդ օրինակը):

Նկատենք նաև, որ չնայած **area()** - ն վիրտուալ ֆունկցիա է, կարողացանք հայտարարել **CPolygon** տիպի օբյեկտ և կանչել նրա **area()** ֆունկցիան, որը միշտ վերադարձնում է **0**:

Աբստրակտ դասեր

Սովորական աբստրակտ դասերը նախորդ օրինակում բերված **CPolygon** դասի նման դասեր են, միայն այն տարբերությամբ, որ նախորդ օրինակում **CPolygon** դասում որոշեցինք **area()** ֆունկցիան, մինչդեռ աբստրակտ դասերում պետք է թողնեինք միայն ֆունկցիայի նախատիպը՝ նրան կցելով **=0** արտահայտությունը: Այսինքն՝ ֆունկցիան չենք որոշում:

Այսպիսով, **CPolygon** դասը կստանա հետևյալ տեսքը՝

```

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
};

```


Ֆունկցիան, որը չենք որոշում (մարմնի փոխարեն գրում ենք =0) կոչվում է **մաքուր վիրտուալ ֆունկցիա**: Ցանկացած դաս, որը պարունակում է **մաքուր վիրտուալ ֆունկցիա**, կոչվում է **աբստրակտ դաս**:

Աբստրակտ դասի կարևորագույն հատկությունն այն է, որ համապատասխան տիպի օբյեկտներ ստեղծվել չեն կարող: Չնայած սրան՝ կարող ենք ստեղծել աբստրակտ դասի ցուցիչներ: Այսպիսով ստացվում է, որ հետևյալ հայտարարությունը՝

```
CPolygon poly;
```

սխալ է, եթե **CPolygon** – ը աբստրակտ դաս է: Իսկ հետևյալ հայտարարությունները ճիշտ են.

```
CPolygon * ppoly1;  
CPolygon * ppoly2;
```

Սա տեղի է ունենում այն պատճառով, որ դասում պարունակվող մաքուր վիրտուալ ֆունկցիան որոշված չէ, իսկ դասի օբյեկտներ կարելի է ստեղծել միայն այն ժամանակ, երբ նրա բոլոր անդամները որոշված են: Սակայն կարող ենք հայտարարել աբստրակտ դասի ցուցիչ, որով հնարավոր կլինի դիմել տվյալ մաքուր վիրտուալ ֆունկցիային համապատասխան՝ ժառանգող դասում սահմանված ֆունկցիային:

Ասվածը պարզաբանելու համար բերենք օրինակ:

```

// virtual andamner
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
};

class CRectangle: public CPolygon {
public:
    int area (void)
        { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
        { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    return 0;
}

```

20
10

Օրինակում կարողացանք **CPolygon*** տիպի ցուցիչով դիմել նաև **CRectangle**, և **CTriangle** տիպի օբյեկտների: Սա շատ օգտակար բան է: Հիմա փոձենք ստեղծել **CPolygon** դասի մեթոդ, որը էկրանին կտպի **area()** ֆունկցիայի արդյունքը: Նկատենք, որ **area()** ֆունկցիան յուրաքանչյուր ժառանգող դասի համար առանձին է սահմանված, իսկ էկրանին տպող ֆունկցիան ցանկանում ենք սահմանել հենց **CPolygon** դասում:

```

// virtual andamner
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
        { cout << this->area() << endl; }
};

class CRectangle: public CPolygon {
public:
    int area (void)
        { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
        { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    return 0;
}

```

20
10

Հիշեք, որ **this** - ը ցույց է տալիս այն օբյեկտը, որի ծրագիրը (կոդը) աշխատում է ներկա պահին:

Բաժին 5.1

Կաղապարներ (Templates)

Ֆունկցիաների կաղապարներ

Կաղապարները հնարավորություն են տալիս ստեղծելու ընդհանուր ֆունկցիաներ, որոնք կարող են աշխատել ցանկացած տիպի տվյալների հետ: Նույն առաջադրանքն առանց կաղապարների իրագործելու համար ստիպված կլինեինք ստեղծել գերբեռնված ֆունկցիաներ՝ բոլոր հնարավոր տիպերի համար: Ֆունկցիայի կաղապար սահմանելու գրելաձևը հետևյալն է.

```
template <class նույնարկիչ> ֆունկցիայի_հայտարարություն;  
template <typename նույնարկիչ> ֆունկցիայի_հայտարարություն;
```

Այս երկու նախատիպերի միջև եղած միակ տարբերությունը **class** կամ **typename** բանալի-բառերի օգտագործումն է: Դրանք երկուսն էլ աշխատում են ճիշտ նույն կերպ և ունեն ճիշտ նույն հատկությունները:

Օրինակ՝ որպեսզի ստեղծենք ֆունկցիայի կաղապար, որը կվերադարձնի իրեն որպես արգումենտ տրված երկու օբյեկտներից մեծագույնը, կարող ենք գրել.

```
template <class MyType>  
MyType GetMax (MyType a, MyType b) {  
    return (a>b?a:b);  
}
```

Առաջին տողում ստեղծում ենք տվյալների տիպի կաղապար, որն անվանում ենք **MyType**: Դրան հաջորդող ֆունկցիայում **MyType** - ը դառնում է լիովին վավեր տիպ: Այդ տիպն օգտագործում ենք որպես **a** և **b** փոփոխականների տիպ, ինչպես նաև **GetMax** ֆունկցիայի վերադարձվող արժեքի տիպ:

MyType - ը, սակայն, ոչ մի կոնկրետ տիպ չի ներկայացնում: **GetMax** ֆունկցիան կարող ենք կանչել ցանկացած վավեր տվյալների տիպով: Այդ ժամանակ **MyType** - ը կփոխարինվի համապատասխան տիպով: Կաղապարային ֆունկցիաները կանչվում են հետևյալ կերպ՝

ֆունկցիա <տիպեր> (արգումենտներ);

Օրինակ՝ որպեսզի կանչենք **GetMax** ֆունկցիան և համեմատենք երկու **int** արժեքներ, կարող ենք գրել

```
int x,y;  
GetMax <int> (x,y);
```

Սրանով **GetMax** ֆունկցիան կկանչվի՝ իրեն դրսևորելով ճիշտ այնպես, ինչպես կդրսևորեր, եթե **MyType** - ի փոխարեն բոլոր տեղերում գրեինք **int** :

Ահա մի օրինակ.

```

// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
}

int main () {
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}

```

6
10

Այս դեպքում հայտարարված ընդհանուր տիպին տվեցինք **T** անուն՝ **MyType** - ի փոխարեն, քանզի այն ավելի կարճ է և կադապարներում ամենաշատն օգտագործվող նույնարկիչն է: Դրա փոխարեն կարող էինք օգտագործել ցանկացած վավեր նույնարկիչ:

Վերևում բերված օրինակում միննույն **GetMax()** ֆունկցիան օգտագործեցինք **int** և **long** տիպի փոփոխականներով:

Ինչպես տեսնում եք, մեր **GetMax()** ֆունկցիայի կադապարում **T** տիպը կարող է օգտագործվել նոր օբյեկտներ հայտարարելու համար.

```
T result;
```

result - ը **T** տիպի օբյեկտ է, ինչպես **a** - ն և **b** - ն: Դա այն տիպն է, որը կգրենք <> նշանների միջև՝ ֆունկցիան կանչելիս:

Կոնկրետ այս դեպքում (երբ **T** տիպն օգտագործվում է որպես **GetMax** ֆունկցիայի արգումենտ) թարգմանիչը կարող է ինքնուրույն որոշել, թե **T** տիպը ինչով պիտի փոխարինվի՝ առանց նշելու **<int>** կամ **<long>** պարամետրերը: Այսպիսով՝ կարող ենք գրել

```
int i,j;
GetMax (i,j);
```

Քանի որ **i** -ն, և **j** -ն **int** տիպի են, թարգմանիչի համար պարզ կլինի, որ անհրաժեշտ ֆունկցիան **int** տիպի է: Այս մեթոդը ավելի հաճախ է կիրառվում և տալիս է նույն արդյունքը:

```

// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
    return (a>b?a:b);
}

int main () {
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax(i,j);
    n=GetMax(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}

```

```

6
10

```

Քանի որ օրինակում բերված կաղապարային ֆունկցիան ունի միայն մի տվյալի ընդհանուր տիպ (**class T**), և նրա ընդունած երկու արգումենտներն էլ նույն տիպի են, այդ կաղապարային ֆունկցիան չենք կարող կանչել երկու տարբեր տիպի արգումենտներով.

```

int i;
long l;
k = GetMax (i,l);

```

Սա սխալ կլինի, քանի որ մեր ֆունկցիան ընդունում է նույն տիպի երկու արգումենտ:

Կարող ենք ստեղծել կաղապարային ֆունկցիաներ, որոնք կունենան մեկից ավելի ընդհանուր տիպ: Օրինակ՝

```

template <class T, class U>
T GetMin (T a, U b) {
    return (a<b?a:b);
}

```

Այս դեպքում **GetMin()** ֆունկցիան ընդունում է երկու տարբեր տիպերի արգումենտներ և վերադարձնում է առաջին արգումենտի տիպի (**T**) արժեք: Վերևում բերված հայտարարությունից հետո կարող ենք ֆունկցիան կանչել հետևյալ կերպ.

```

int i,j;
long l;
i = GetMin<int,long> (j,l);

```

կամ պարզապես

```

i = GetMin (j,l);

```

Այս դեպքում ևս թարգմանիչը կարող է ինքնուրույն որոշել թե ինչպիսի ֆունկցիա է պետք կառուցել կադապարի միջոցով:

Դասերի կադապարներ

Կադապարային ֆունցիաների հետ մեկտեղ C++ - ում իրականացված է կադապարային դասերի գաղափարը: Մրանց օգտագործումը շատ նման է կադապարային ֆունկցիաների օգտագործմանը: Օրինակ՝

```
template <class T>
class pair {
    T values [2];
public:
    pair (T first, T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

Այս դասը հնարավորություն է տալիս պահելու ցանկացած գոյություն ունեցող տիպի երկու օբյեկտ: Օրինակ՝ էթե ուզում ենք հայտարարել այս դասի օբյեկտ, որը պիտի պահի երկու **int** տիպի տարր, որոնց արժեքները **115** և **36** են, կարող ենք գրել.

```
pair<int> myobject (115, 36);
```

Իսկ, օրինակ, **float** տիպի զույգ պահելու համար կարող ենք գրել

```
pair<float> myfloats (3.0, 2.18);
```

Վերևում բերված օրինակում միակ անդամ ֆունկցիան գրեցինք դասի հայտարարության մեջ: Եթե ֆունկցիան առանձնացնում ենք դասից, ապա այն ամպայման պիտի սկսի **template <...>** - ուղ:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class pair {
    T value1, value2;
public:
    pair (T first, T second)
        {value1=first; value2=second;}
    T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
    T retval;
    retval = value1>value2? value1 : value2;
    return retval;
}
```

```
int main () {
    pair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
```

100

Կաղապարի պարամետրեր

Կաղապար հայտարարելիս նրան տիպի անուն փոխանցելու համար օգտագործում ենք **class** կամ **typename** բանալի-բառերը: Սակայն կաղապարին կարող ենք փոխանցել նաև ցանկացած հաստատուն: Դիտարկենք հետևյալ օրինակը.

```
// array template
#include <iostream>
using namespace std;

template <class T, int N>
class array {
    T memblock [N];
public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember (int x, T value) {
    memblock[x]=value;
}

template <class T, int N>
T array<T,N>::getmember (int x) {
    return memblock[x];
}

int main () {
    array <int,5> myints;
    array <float,5> myfloats;
    myints.setmember (0,100);
    myfloats.setmember (3,3.1416);
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
    return 0;
}
```

100

3.1416

Փոխանցվող բոլոր պարամետրերի համար կարելի է սահմանել լռության արժեքներ, ինչպես դա կատարվում է ֆունկցիաների արգումենտների համար:

Ահա կաղապարների հայտարարման մի քանի օրինակ՝


```

template <class T> //amenahachax patahox@` miak tipayin parametrov
template <class T, class U> //erku tipayin parametrerov
template <class T, int N> //tipayin parametr yev yev amboxch tiv
template <class T = char> //lrutyany arjeqov
template <int Tfunc (int)> //funkcia` vorpes parametr

```

Կադապարներ և մի քանի ֆայլերով նախագծեր

Թարգմանչի տեսակետից նայելիս կադապարները սովորական դասեր և ֆունկցիաներ չեն. դրանք թարգմանվում են միայն այն դեպքում, երբ կա համապատասխան կանչ: Այսինքն, երբ թարգմանիչը ծրագրի կոդի մեջ գտնում է կադապարային ֆունկցիայի կանչ, այն, հիմնվելով կադապարի վրա, ստեղծում է համապատասխան ֆունկցիա:

Մեծ նախագծեր ստեղծելիս ծրագրի կոդը բաժանում են տարբեր ֆայլերի միջև: Այս դեպքում սովորաբար դասերի/ֆունկցիաների ինտերֆեյսը (հայտարարությունը) և իրագործումը (որոշումը) տարբեր ֆայլերի մեջ են գրվում: Բոլոր ֆունկցիաների նախատիպերը գրվում են **header** ֆայլերի մեջ (սովորաբար՝ **.h** վերջավորությամբ), իսկ ֆունկցիաների որոշումները՝ **C++** կոդի առանձին ֆայլում:

Կադապարների օգտագործման դեպքում այս մոտեցումը կիրառելի չէ: Կադապարային ֆունկցիայի/դասի հայտարարությունը և որոշումը չեն կարող լինել տարբեր ֆայլերում:

Բաժին 5.2

Նախաթարգմանչի հրամաններ (Preprocessor Directives)

Նախաթարգմանչի հրամանները կատարվում են ոչ թե ծրագրի, այլ նախաթարգմանչի կողմից: Նախաթարգմանչին աշխատեցվում է այն ժամանակ, երբ թարգմանչին հրամայում ենք թարգմանել կոդը: Նախաթարգմանչը ստուգում է կոդը, կատարում իրեն տրված հրամանները և սխալ առաջանալու դեպքում կանգնեցնում է թարգմանման պրոցեսը:

Նախաթարգմանչի բոլոր հրամանները գրվում են առանձին տողի վրա, և նրանցից հետո չի դրվում `;` նշանը:

#define

Այս գրքի սկզբում արդեն ծանոթացել ենք նախաթարգմանչի `#define` հրամանին: Դրա միջոցով ստեղծում ենք **նշանակված փոփոխականներ** և **մակրոհրամաններ**: Գրելաձևը հետևյալն է՝

`#define` **անուն** **արժեք**

Նախաթարգմանչը, ամեն անգամ կոդի մեջ հանդիպելով **անուն** բառին, այն փոխարինում է **արժեք** արտահայտությամբ: Օրինակ՝

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
char str2[MAX_WIDTH];
```

Սա հայտարարում է 100 երկարությամբ երկու սիմվոլային տողեր:

`#define` - ի միջոցով կարող ենք նաև հայտարարել մակրո ֆունկցիաներ: Օրինակ՝

```
#define max(a,b) a>b?a:b
int x=5, y;
y = max(x,2);
```

Կոդի այս հատվածի կատարվելուց հետո `y` - ը կպարունակի 5:

#undef

`#undef` - ը կատարում է `#define` - ի հակառակ գործողությունները: Այն տրված արտահայտությունը հանում է նշանակված հաստատունների ցուցակից:

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
#undef MAX_WIDTH
#define MAX_WIDTH 200
char str2[MAX_WIDTH];
```

#ifdef, #ifndef, #if, #endif, #else and #elif

Այս հրամանները հնարավորություն են տալիս ծրագրի կոդի մի մասն անտեսել՝ կախված ինչ որ պայմաններից: Պետք է հիշել, որ սա կատարվում է ծրագիրը թարգմանելիս, այլ ոչ թե աշխատեցնելիս:

#ifdef - ը հնարավորություն է տալիս ծրագրի մի մասը թարգմանել միայն այն դեպքում, երբ այն անունը, որը տրվել է նրան որպես արգումենտ, նշանակված հաստատուն է (հաստատունի արժեքը այստեղ նշանակություն չունի): Գրելաձևը հետևյալն է՝

```
#ifdef անուն
// ծրագրի կոդը գրվում է այստեղ
#endif
```

Օրինակ՝

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

Այս դեպքում **char str[MAX_WIDTH];** տողին թարգմանիչն «ուշադրություն է դարձնում» միայն այն դեպքում, երբ **MAX_WIDTH** - ը նշանակված հաստատուն է: Եթե այն նշանակված հաստատուն չէ, այդ տողը անտեսվում է (ծրագրի մեջ չի ներառվում):

#ifndef - ը ծառայում է հակառակ նպատակին: **#ifndef** - ի և **#endif** - ի միջև գրված կոդը թարգմանվում է միայն այն դեպքում, երբ նշված անունը նշանակված հաստատուն չէ: Օրինակ՝

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 100
#endif
char str[MAX_WIDTH];
```

Եթե նախաթարգմանիչը հասել է կոդի այս հատվածին, և **MAX_WIDTH** - ը նշանակված հաստատուն չէ, այն կնշանակվի, և նրան կտրվի 100 արժեք: Եթե այն արդեն գոյություն ունի, ապա կպահպանի իր արժեքը (քանի որ **#define** հրամանը չի աշխատեցվի):

#if, #else և **#elif** (**elif**՝ կրճատած **else if** - ից) հրամաններից հետո գրված կոդը թարգմանվում է միայն որոշակի պայմաններում: Սրանք կարող են աշխատել միայն **նշանակված հաստատունների** հետ: Օրինակ՝

```
#if MAX_WIDTH>200
    #undef MAX_WIDTH
    #define MAX_WIDTH 200
#elif MAX_WIDTH<50
    #undef MAX_WIDTH
    #define MAX_WIDTH 50
#else
    #undef MAX_WIDTH
```

```
#define MAX_WIDTH 100
#endif

char str[MAX_WIDTH];
```

Ուշադրություն դարձնենք, թե ինչպես է **#if**, **#elsif** և **#else** շարքը ավարտվում **#endif** - ով:

#line

Երբ թարգմանչին հրամայում ենք թարգմանել ծրագիրը, և թարգմանման ընթացքում որևէ սխալ է առաջանում, թարգմանիչը ցույց է տալիս այն ֆայլի անունը և տողի համարը, որտեղ տեղի է ունեցել սխալը:

#line հրամանը հնարավորություն է տալիս փոփոխելու երկուսն էլ՝ տողերի համարակալումը և ֆայլի անունը, որոնք ցույց կտրվեն սխալ առաջանալու դեպքում: Գրելաձևը հետևյալն է՝

```
#line տողի_համար "ֆայլի_անուն"
```

Այստեղ **տողի_համարն** այն համարն է, որը կտրվի կողի հաջորդ տողին: Հաջորդող տողերը կհամարակալվեն՝ սկսած այդ համարից:

Ֆայլի_անունը ոչ պարտադիր արգումենտ է: Եթե այն նշված է, ապա ծրագիրը թարգմանելիս սխալներ առաջանալու դեպքում ցույց կտրվի ոչ թե իրական ֆայլի անունը, այլ **#line** հրամանով նշվածը: Օրինակ՝

```
#line 1 "haytararum enq popoxakanner"
int a?;
```

Այս ծրագիրը թարգմանելիս թարգմանիչը սխալ ցույց կտա "haytararum enq popoxakanner" ֆայլի 1 տողում:

#error

Այս հրամանին հանդիպելիս նախաթարգմանիչը կանգնեցնում է թարգմանման պրոցեսը՝ որպես սխալ վերադարձնելով իրեն տրված տողը.

```
#ifndef __cplusplus
#error C++ - i targmanich chka
#endif
```

Այս օրինակը կանգնեցնում է թարգմանման պրոցեսը, եթե **__cplusplus** - ը նշանակված հաստատուն չէ:

#include

Այս հրամանը նույնպես բազմիցս օգտագործվել է այս գրքում: Երբ նախաթարգմանիչը հանդիպում է **#include** հրամանին, այն փոխարինում է այդ

տողը նշված ֆայլի պարունակությամբ (իրականում այդ պրոցեսը փոքր ինչ ավելի բարդ է, բայց արդյունքը նույնն է): Կա կցվելիք ֆայլը նշելու երկու եղանակ.

```
#include "ֆայլ"
```

```
#include <ֆայլ>
```

Այս երկու ձևերի տարբերությունն այն է, որ առաջին դեպքում նախաթարգմանիչը նշված ֆայլը փնտրում է նույն պանակում (folder), որտեղ գտնվում է ներկայումս թարգմանվող ֆայլը: Մյուս դեպքում նախաթարգմանիչը ֆայլը փնտրում է հատուկ պանակներում, որտեղ գտնվում են ստանդարտ գրադարանները:

```
#pragma
```

Սա օգտագործվում է թարգմանչին հատուկ հրամաններ տալու համար: Այդ հրամանները տարբեր թարգմանիչների համար տարբեր են: Դրանց ծանոթանալու համար կարդացեք ձեր թարգմանչի ձեռնարկը:

Բաժին 6.1

Մուտք/Ելք ֆայլերի հետ

C++ - ը հնարավորություն է տալիս ֆայլերի հետ մուտք/ելք կատարել՝ օգտագործելով հետևյալ դասերը՝

- **ofstream**՝ ֆայլերի մեջ գրելու համար (ժառանգված է **ostream** - ից)
- **ifstream**՝ ֆայլերից կարդալու համար (ժառանգված է **istream** - ից)
- **fstream**՝ ն՝ գրելու, ն՝ կարդալու համար (ժառանգված է **iostream** - ից)

Ֆայլի բացում

Այս դասերի օբյեկտների հետ սովորաբար առաջին գործողությունը այն կապելն է որևէ իրական ֆայլի հետ. այլ խոսքով ասած՝ ֆայլ բացելը: Բացված ֆայլը ներկայացվում է որպես հոսքի օբյեկտ (ինչպես **cout** - ը և **cin** - ը), և այդ հոսքի հետ կատարվող ցանկացած մուտք/ելք կատարվում է նաև ֆայլի հետ:

Որևէ ֆայլ բացելու համար օգտագործում ենք հոսքի օբյեկտի **open()** մեթոդը.

```
void open (const char * ֆայլի անուն, openmode բացման ձև);
```

Այստեղ **ֆայլի անունը** սիմվոլային տող է, որը ներկայացնում է ֆայլի անունը և, անհրաժեշտության դեպքում, հասցեն ("C:\file.txt"): Իսկ **բացման ձևը** հետևյալ դրոշակների որևէ համակցում է՝

ios::in	ֆայլը բացել կարդալու համար
ios::out	ֆայլը բացել գրելու համար
ios::ate	սկզբնական դիրքը՝ ֆայլի վերջում
ios::app	բոլոր փոփոխությունները գրվում են՝ սկսած ֆայլի վերջից
ios::trunc	եթե ֆայլը գոյություն ունի, այն մաքրվում է
ios::binary	երկուական ռեժիմ

Այս դրոշակները կարելի է համակցել (միանգամից ընտրել մի քանսը)՝ օգտագործելով բիթային ԿԱՄ օպերատորը՝ **|**: Օրինակ՝ եթե ցանկանում ենք բացել **orinak.bin** ֆայլը և նրա մեջ սվյալներ գրել երկուական ռեժիմում, ապա պետք է կանչենք **open** ֆունկցիան հետևյալ կերպ՝

```
ofstream file;  
file.open ("orinak.bin", ios::out | ios::app | ios::binary);
```

Բոլոր երեք դասերի (**ofstream**, **ifstream** and **fstream**) **open** մեթոդներն ունեն լրության **բացման ձև**: Դասերից յուրաքանչյուրի համար այն յուրահատուկ է.

դաս	Լռության բացման ձև
ofstream	<code>ios::out ios::trunc</code>
ifstream	<code>ios::in</code>
fstream	<code>ios::in ios::out</code>

Լռության արժեքն օգտագործվում է միայն այն դեպքում, երբ **open** մեթոդը կանչելիս չի նշվում բացման ձև. հակառակ դեպքում լռության արժեքն անտեսվում է:

Ինչպես ասացինք, առաջին գործողությունը, որ արվում է **ofstream**, **ifstream** և **fstream** դասերի օբյեկտների հետ, ֆայլ բացելն է. այդ պատճառով այս դասերն ունեն կառուցիչ, որը միանգամից կանչում է **open** մեթոդը: Այս ձևով կարող էինք հայտարարել նախորդ օբյեկտը՝ պարզապես գրելով

```
ofstream file ("orinak.bin", ios::out | ios::app | ios::binary);
```

Երկու եղանակն էլ ունեն նույն ազդեցությունը:

Կառող ենք ստուգել՝ արդյոք ֆայլի բացման պրոցեսը նորմալ է անցել՝ օգտագործելով **is_open()** մեթոդը.

```
bool is_open();
```

Սա վերադարձնում է բուլյան տիպ՝ **true**, եթե օբյեկտը նորմալ կապվել է ֆայլին, իսկ հակառակ դեպքում՝ **false**:

Ֆայլի փակում

Ֆայլը բացելուց և նրա հետ աշխատելուց հետո այն պետք է փակել, որպեսզի այն կրկին հասանելի դառնա մյուս ծրագրերի համար: Դա անելու համար պետք է կանչել **close()** մեթոդը, որը ավարտում է ֆայլի հետ աշխատանքը և փակում ֆայլը: **close()** – ի նախատիպը հետևյալն է՝

```
void close ();
```

Այս ֆունկցիան կանչելուց հետո հոսքի օբյեկտը կարելի է օգտագործել այլ ֆայլեր բացելու համար, իսկ փակված ֆայլը դառնում է այլ ծրագրերի համար հասանելի:

Եթե հոսքի օբյեկտը ոչնչացվում է (օրինակ՝ եթե ծրագիրն ավարտվում է), իսկ ֆայլը դեռ բաց է, փլուփզիչը ավտոմատ կանչում է **close()** մեթոդը:

Ֆայլերի հետ աշխատանք տեքստային ռեժիմում

ofstream, **ifstream** և **fstream** դասերը համապատասխանաբար ժառանգված են **ostream**, **istream** և **iostream** դասերից: Դրա շնորհիվ կարող ենք **fstream** օբյեկտների հետ աշխատել ձևով դասերի ֆունկցիաներով:

Տեքստային ռեժիմում ֆայլերի հետ աշխատելիս **ofstream**, **ifstream** և **fstream** դասերի օբյեկտների հետ կաշխատենք այնպես, ինչպես **cin** - ի և **cout** - ի հետ: Ստորև բերված է << օպերատորի օգտագործման օրինակ:

```
// greng textayin file-i mech
#include <fstream.h>
int main () {
    ofstream orinakfile ("orinak.txt");
    if (orinakfile.is_open()) {
        orinakfile << "Sa mi tox e.\n";
        orinakfile << "Sa mek ayl tox e.\n";
        orinakfile.close();
    }
    return 0;
}
```

file orinak.txt

Sa mi tox e.
Sa mek ayl tox e.

Տվյալների մուտքը կազմակերպում ենք ճիշտ այնպես, ինչպես **cin** - ի հետ.

```
// kardang textayin file
#include <iostream>
using namespace std;
#include <fstream.h>
#include <stdlib.h>

int main () {
    char buffer[256];
    ifstream orinakfile ("example.txt");
    if (! orinakfile.is_open())
        { cout << "Sxal file-@ bacelis"; exit (1); }

    while (! orinakfile.eof() )
    {
        orinakfile.getline (buffer,100);
        cout << buffer << endl;
    }
    return 0;
}
```

Sa mi tox e.
Sa mek ayl tox e.

Այս օրինակը կարդում է տեքստային ֆայլը և նրա պարունակությունը տպում էկրանին: Այստեղ օգտագործեցինք մի նոր մեթոդ՝ **eof**: Այս մեթոդը **ifstream** - ը ժառանգել է **ios** դասից: Սա վերադարձնում է **true**, եթե հասել ենք ֆայլի վերջին:

Վիճակի դրոշակների ստուգում

Բացի `eof()` – ից կան նաև այլ անդամ-ֆունկցիաներ, որոնց միջոցով կարելի է տեղեկանալ հոսքի վիճակի մասին (այս բոլոր ֆունկցիաները վերադարձնում են `bool` արժեք)։

`bad()`

Վերադարձնում է `true`, եթե ֆայլը կարդալիս կամ գրելիս որևէ խնդիր է առաջացել։ Օրինակ՝ եթե ֆայլը բացված է միայն կարդալու համար, իսկ մենք գրելու փորձ ենք կատարում, կամ եթե այն սարքը, որի վրա ուզում ենք գրել, ամբողջովին զբաղված է կամ գրելուց պաշտպանված։

`fail()`

Վերադարձնում է `true` բոլոր այն դեպքերում, երբ `true` է վերադարձնում `bad()` - ը, և այն դեպքերում, երբ առաջանում է տվյալների ֆորմատավորման սխալ (օրինակ՝ ծրագիրը փորձում է որևէ թիվ կարդալ, բայց ստանում է տեքստ)։

`eof()`

Վերադարձնում է `true`, երբ կարդալու համար բացված ֆայլը հասել է վերջին (այլևս կարդալու սիմվոլ չկա)։

`good()`

Վերադարձնում է `false` բոլոր այն դեպքերում, երբ նախորդներից որևէ մեկը կվերադարձնե `true`։

`get` և `put` հոսքային ցուցիչներ

Բոլոր մուտքի-ելքի օբյեկտներն ունեն գոնե մեկ հոսքային ցուցիչ։

- `ifstream` - ը `istream` - ի պես ունի ցուցիչ, որը կոչվում է `get pointer`, որը ցույց է տալիս հաջորդ կարդացվելիք տարրը։
- `ofstream` - ը `ostream` - ի պես ունի ցուցիչ, որը կոչվում է `put pointer`, որը ցույց է տալիս այն տեղը, որտեղ պետք է գրվի հաջորդ տարրը։
- Եվ վերջապես՝ `fstream` - ը `iostream` - ի պես ունի և՛ `get`, և՛ `put` ցուցիչներ։

Այս ցուցիչները կարող ենք օգտագործել հետևյալ անդամ-ֆունկցիաների միջոցով.

`tellg()` և `tellp()`

Այս երկու ֆունկցիաները արգումենտներ չեն պահանջում և վերադարձնում են `pos_type` տիպի արժեք, որը ամբողջ թիվ է և իրենից ներկայացնում է համապատասխանաբար, `get` և `put` հոսքային ցուցիչների ներկա դիրքերը։

seekg() և seekp()

Մրանք օգտագործվում են **get** և **put** հոսքային ցուցիչների դիրքերը փոխելու համար: Երկուսն էլ գերբեռնված են երկու տարբեր նախատիպերով:

```
seekg ( pos_type դիրք );  
seekp ( pos_type դիրք );
```

Այս դեպքում տալիս ենք ցուցիչի դիրքը՝ հաշված ֆայլի սկզբից: **Դիրքը** պետք է ունենա նույն տիպը, ինչ վերադարձվում է **tellg** և **tellp** անդամ-ֆունկցիաների կողմից:

```
seekg ( off_type դիրք, seekdir ուղղություն );  
seekp ( off_type դիրք, seekdir ուղղություն );
```

Այս ֆունկցիաներն օգտագործելիս կարող ենք նշել, թե որտեղից հաշվել տրված **դիրքը**: **Ուղղությունը** կարող է լինել հետևյալներից որևէ մեկը.

ios::beg	հաշված հոսքի (ֆայլի) սկզբից
ios::cur	հաշված ցուցիչի ներկա դիրքից
ios::end	հաշված հոսքի վերջից

Հաջորդ օրինակն օգտագործում է այս ֆունկցիաները՝ որոշելու համար երկուական ֆայլի չափը:

```
// obtaining file size  
#include <iostream>  
using namespace std;  
#include <fstream.h>  
const char * filename = "orinak.txt";  
int main ()  
{  
    long l,m; ifstream  
    file (filename, ios::in|ios::binary);  
    l = file.tellg();  
    file.seekg (0, ios::end);  
    m = file.tellg();  
    file.close();  
    cout << filename << "-i chap@ ";  
    cout << (m-l) << " byte e.\n";  
    return 0;  
}
```

```
orinak.txt-i chap@ 33 byte e.
```

Երկուական ֆայլեր

Երկուական ֆայլերի մեջ տվյալների մուտքը/ելքը << և >> օպերատորների ինչպես նաև **getline** ֆունկցիայի միջոցով անհիմաստ է, սակայն դրանք լիովին թույլատրելի գործողություններ են:

Հոսքերն ունեն երկու հատուկ ֆունկցիաներ մուտքի/ելքի համար. դրանք են՝ **write** և **read**: Դրանցից առաջինը (**write**) **ostream** դասի անդամ-ֆունկցիա է և ժառանգված է **ofstream** - ի կողմից, իսկ **read** - ը **istream** դասի անդամ է և ժառանգված է **ifstream** - ի կողմից: **fstream** դասի օբյեկտներն ունեն և՛ **write**, և՛ **read** ֆունկցիաները: Այդ ֆունկցիաների նախատիպերը հետևյալն են՝

```
write ( char * buffer, streamsize չափ );
read ( char * buffer, streamsize չափ );
```

Այստեղ **buffer** - ը հիշողության հասցե է, որտեղ գրվում են կարդացված տվյալները, և որտեղից կարդացվում է գրվելիք ինֆորմացիան: Իսկ **չափ** արգումենտն ամբողջ թիվ է, որը ներկայացնում է **buffer** - ից(ում) կարդացվելիք/գրվելիք սիմվոլների քանակը:

```
// kardum enq erkuakan file
#include <iostream>
using namespace std;
#include <fstream.h>
const char * filename = "example.txt";
int main ()
{
    char *
    buffer;
    long size;
    ifstream file (filename, ios::in|ios::binary|ios::ate);
    size = file.tellg();
    file.seekg (0, ios::beg);
    buffer = new char [size];
    file.read (buffer, size);
    file.close();
    cout << "ayjm file-@ amboxchovin gtnvum e buffer-um";

    delete[] buffer;
    return 0;
}
ayjm file-@ amboxchovin gtnvum e buffer-um
```

Բուֆերներ և Սինխրոնիզացիա

Ֆայլային հոսքերը կապված են **streambuf** տիպի բուֆերի հետ: Բուֆերը հիշողության բլոկ է, որը աշխատում է որպես միջնորդ՝ հոսքի և ֆիզիկական ֆայլի միջև: Օրինակ՝ ելքի հոսքի դեպքում ամեն անգամ **put** անդամ-ֆունկցիան կանչելիս (մի սիմվոլ գրելու համար) սիմվոլը միանգամից չի գրվում ֆայլի մեջ. փոխարենը այն տեղադրվում է բուֆերի մեջ:

Երբ բուֆերը դատարկվում է (**flush**), նրա պարունակությունը գրվում է ֆայլի մեջ (եթե դա ելքի հոսք է) կամ ջնջվում է (մուտքի հոսքի դեպքում): Այս պրոցեսը կոչվում է սինխրոնիզացիա և տեղի է ունենում հետևյալ դեպքերում:

- **Երբ ֆայլը փակվում է.** ֆայլը փակելուց առաջ բոլոր բուֆերները սինխրոնիզացվում են:

- **Երբ բուֆերը լցվում է.** բուֆերներն ունեն որոշակի չափեր, և երբ բուֆերն ամբողջովին լցվում է, այն սինխրոնիզացվում է:
- **Հատուկ դեպքերում.** `flush` և `endl` ազդակներն օգտագործելիս բուֆերը սինխրոնիզացվում է:
- **`sync()` ֆունկցիայի կանչի դեպքում.** `sync()` (արգումենտներ չկան) անդամ-ֆունկցիայի կանչը բերում է անհապաղ սինխրոնիզացիայի: Այս ֆունկցիան վերադարձնում է `int` տիպի արժեք, որը հավասար է (-1) - ի, եթե հոսքին ոչ մի բուֆեր կապած չէ կամ եթե որևէ խնդիր է առաջացել:

Բաժին 7.1

Անվանատարածքներ (namespaces)

Անվանատարածքները թույլ են տալիս տարբեր տարրեր՝ դասեր, օբյեկտներ, ֆունկցիաներ, համախմբել մեկ անվան տակ: Մրանով դուք կարող եք գլոբալ տեսանելիության տիրույթը բաժանել ենթա-տիրույթների՝ յուրաքանչյուրը սեփական անունով:

Անվանատարածք հայտարարելու նախատիպը հետևյալն է.

```
namespace անուն
{
    պարունակություն
}
```

որտեղ *անունը*՝ անվանատարածքի անունն է, իսկ *պարունակությունը*՝ դասերի, օբյեկտների և ֆունկցիաների բազմությունն է: Օրինակ՝

```
namespace ImTaracq
{
    int a, b;
}
```

Այս դեպքում `a`-ն և `b`-ն սովորական փոփոխականներ են՝ հայտարարված `ImTaracq` անվանատարածքում: Այդ անվատարածքից դուրս դրանց դիմելու համար անհրաժեշտ է օգտագործել տեսանելիության տիրույթի `::` օպերատորը՝

```
ImTaracq::a
ImTaracq::b
```

Անվանատարածքները հիմնականում օգտագործվում են նույն անունով գլոբալ օբյեկտները կամ ֆունկցիաները իրարից տարբերելու համար: Օրինակ պատկերացրեք, որ դուք հայտարարել եք գլոբալ փոփոխական, որի անունն արդեն օգտագործված է ներառվող գրադարանի մեջ: Դա կբերի կոմպիլացիայի սխալների: Այդ պատճառով ընդունված է գրադարաններ գրելիս, դրանք զետեղել ինչ-որ անվանատարածքի մեջ, ինչպես և արված է `iostream` գրադարանում, որը զետեղված է `std` անվանատարածքի մեջ: Օրինակ՝

```

// anvanataracqner

#include <iostream>
using namespace std;

namespace ImTaracq1
{
    int a = 5;
}

namespace ImTaracq2
{
    double a = 2.5;
}

void main() {
    cout << ImTaracq1::a << endl;
    cout << ImTaracq2::a << endl;
}

```

```

5
2.5

```

Այս դեպքում մենք ունենք նույն անունով երկու փոփոխականներ՝ a: Առաջինը հայտարարված է ImTaracq1 անվանատարածքի մեջ, իսկ երկրորդը՝ ImTaracq2: Եվ անվանատարածքների շնորհիվ մենք չենք ստանում որևէ ստալ:

using

Այս բանալի-բառը օգտագործվում է նշված անվանատարածքից որևէ անուն տվյալ տիրույթին ներկայացնելու համար: Օրինակ,

```

// anvanataracqner

#include <iostream>
using namespace std;

namespace ImTaracq1
{
    int a = 5;
    int b = 10;
}

namespace ImTaracq2
{
    double a = 2.5;
    double b = 7.4;
}

void main() {
    using ImTaracq1::a;
    using ImTaracq2::b;
    cout << a << endl;
    cout << b << endl;
}

```

```
cout << ImTaracq1::b << endl;
cout << ImTaracq2::a << endl;
}
```

5
7.4
10
2.5

Ուշադրություն դարձրեք, թե ինչպես այս ծրագրում *a* փոփոխականը վերաբերում է *ImTaracq1::a*-ն, իսկ *b*-ն՝ *ImTaracq2::b*-ն: Սակայն *ImTaracq1::b*-ն ու *ImTaracq2::a*-ն դիմելու համար մենք պետք է օգտագործենք իրենց լրիվ անունները:

Սակայն մենք կարող ենք *using* բանալի-բառը օգտագործվել ոչ միայն անվանատարածքի որևէ անուն ուղիղ ձևով հասանելի դարձնելու համար, այլ նաև նշված անվանատարածքի բոլոր անունները ուղիղ հասանելի դարձնելու համար: Դրա համար մենք պետք է գրենք այսպես.

```
using namespace անուն ;
```

որտեղ *անունը*՝ այն անվանատարածքի անունն է, որի տարրերը դուք ուզում եք դարձնել տեսանելի: Օրինակ.

```
// anvanataracqner
#include <iostream>
using namespace std;

namespace ImTaracq1
{
    int a = 5;
    int b = 10;
}

namespace ImTaracq2
{
    double a = 2.5;
    double b = 7.4;
}

void main() {
    {
        using namespace ImTaracq1;
        cout << a << endl;
        cout << b << endl;
    }
    {
        using namespace ImTaracq2;
        cout << a << endl;
        cout << b << endl;
    }
}
```

5
10

2.5
7.4

Այս օրինակում մենք `main` ֆունկցիայի մեջ հայտարարել ենք երկու տիրույթ, որոնցից առաջինում տեսանելի ենք դարձրել `ImTaracq1` անվանատարածքի տարրերը, իսկ երկրորդում՝ `ImTaracq1` անվանատարածքի տարրերը:

Անվանատարածքների այլընտրանքային անուններ

Մենք կարող ենք որոշված անվանատարածքներին տալ մեկ այլ անուն հետևյալ կերպ.

```
namespace նոր_անուն = գործող_անուն ;
```

Օրինակ նախորդ օրինակում մենք կարող ենք `ImTaracq1`-ն տալ մեկ այլ անուն՝ `ImLavTaracq`

```
namespace ImLavTaracq = ImTaracq1;
```

std անվանատարածք

C++ -ի ստանդարդ գրադարանների բոլոր տարրերը զետեղված են `std` անունով անվանատարածքի մեջ: Այդ է պատճառը, որ մեր բոլոր ծրագրում գրված է `using namespace std`-ն, քանզի մենք ամենուրեք օգտագործում ենք `iostream` ստանդարդ գրադարանը: