

# C++-ի

## Հիմունքները

Հեղինակներ՝  
Ռուբեն Վարդանյան  
Սուրեն Կարապետյան

Սրբազրիչ՝  
Պետրոս Հարությունյան

Գիրքը հիմնված է ժուռնալ Սուրենի C++-ի նյութերի վրա  
© 2005 Ռ.Ն. Վարդանյան և Ս.Գ.Կարապետյան

# Բովանդակություն

Բովանդակություն .....	2
Ներածություն .....	5
Թարգմանիչներ .....	5
Console ծրագրերի պրոյեկտներ Microsoft Visual C++ 6 -ով .....	5
Նոր պրոյեկտի ստեղծում .....	5
Պրոյեկտի կոմպիլացիա (compilation) և աշխատեցում (execution) .....	8
Բաժին 1.1 C++-ի կառուցվածքը .....	9
Մեկնաբանություններ (Comments) .....	11
Բաժին 1.2 Փոփոխականներ: Ինֆորմացիայի տիպեր: Հաստատուններ .....	12
Նույնարկիչներ (Identifiers) .....	12
Ինֆորմացիայի տիպեր (Data types) .....	13
Փոփոխականների հայտարարումը (Declaration of variables) .....	14
Փոփոխականների սկզբնաթեքավորում (Initialization of variables) .....	15
Փոփոխականների տեսանելիության տիրույթ (Scope of variables) .....	16
Հաստատուններ (Constants) .....	17
Ամբողջ թվեր (Integer Numbers) .....	17
Լողացող կետով թվեր (Floating Point Numbers) .....	17
Սիմվոլներ և տողեր (Characters and strings) .....	17
Նշանակված հաստատուններ (Defined constants) ( <i>#define</i> ) .....	19
Հայտարարված հաստատուններ (Declared constants) ( <i>const</i> ) .....	19
Բաժին 1.3 Օպերատորներ .....	20
Վերագրում (=) .....	20
Թվաբանական գործողություններ (+, -, *, /, %)	21
Բարդ վերագրման օպերատորներ (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=,  =) .....	21
Մեծացման և փոքրացման օպերատորներ .....	21
Համեմատության օպերատորներ (==, !=, >, <, >=, <=) .....	22
Տրամաբանական օպերատորներ ( !, &&,    ) .....	23
Պայմանական օպերատոր ( ? ) .....	24
Բիթային օպերատորներ (&,  , ^, ~, <<, >>) .....	24
Տիպերի ձևափոխման օպերատորներ .....	24
sizeof() .....	25
Օպերատորների նախապատվությունը .....	25
Բաժին 1.4 Հաղորդակցություն օգտագործողի հետ .....	27
Ելք (Output) ( <i>cout</i> ) .....	27
Մուտք (Input) ( <i>cin</i> ) .....	28
Բաժին 2.1 Կառավարման ստրուկտուրաներ (համակարգեր) (Control structures) .....	30
Պայմանային ստրուկտուրա (համակարգ) <i>if</i> և <i>else</i> (Conditional structure: <i>if</i> and <i>else</i> ) .....	30
Կրկնվող ստրուկտուրաներ (համակարգեր) կամ ցիկլեր (Repetitive structures or loops) .....	31
<i>while</i> ցիկլը .....	31
<i>do-while</i> ցիկլը .....	32
<i>for</i> ցիկլը .....	33
Անցման հրամաններ (Jumps) .....	34
<i>break</i> հրամանը .....	34
<i>continue</i> հրամանը .....	35
<i>goto</i> հրամանը .....	35
Ընտրության ստրուկտուրա` <i>switch</i> .....	35
Բաժին 2.2 Ֆունկցիաներ ( Functions ) (I) .....	38

Ֆունկցիաներ՝ առանց տեսակի: <i>void</i> -ի օգտագործումը .....	41
Բաժին 2.3 Ֆունկցիաներ (Functions) (II) .....	43
Արգումենտների փոխանցումը արժեքով ( <i>by value</i> ) և .....	43
հասցեով ( <i>by reference</i> ) .....	43
Արգումենտների լռության (Default) արժեքներ .....	45
Ֆունկցիաների ծանրաբեռնում (Overloading) .....	45
<i>inline</i> ( <i>տողամիջյան</i> ) ֆունկցիաներ .....	46
Դեկուրսիա (Recursivity) .....	47
Ֆունկցիայի նախատիպ (Prototype) .....	47
Բաժին 3.1 Չանգվածներ (Arrays) .....	50
Չանգվածների սկզբնարժեքավորումը (Initialization of arrays) .....	50
Չանգվածների էլեմենտներին դիմումը (Access to the values of an array) .....	51
Բազմաչափ զանգվածներ (Multidimensional Arrays) .....	52
Չանգվածները, որպես պարամետրեր .....	54
Բաժին 3.2 Միավորային տողեր (Strings of characters) .....	56
Տողերի սկզբնարժեքավորում .....	56
Արժեքների վերագրումը տողերին .....	57
Տողերի կոնվերտացիան ուրիշ տիպերի .....	60
Բաժին 3.3 Ցուցիչներ (Pointers) .....	61
Հասցեավորման (Address) օպերատոր (&) .....	61
Ետհասցեավորման (Reference) օպերատոր (*) .....	61
«Ցուցիչ» տիպի փոփոխականների հայտարարում .....	63
Ցուցիչներ և զանգվածներ .....	64
Ցուցիչների սկզբնարժեքավորում (Initialization) .....	65
Ցուցիչների թվաբանությունը .....	66
Ցուցիչ ցուցիչի վրա .....	68
<i>void</i> ցուցիչներ .....	68
Ցուցիչ ֆունկցիայի վրա .....	69
Բաժին 3.4 Դինամիկ հիշողություն (Dynamic memory) .....	70
<i>new</i> և <i>new []</i> օպերատորները .....	70
<i>delete</i> օպերատորը .....	71
Բաժին 3.5 Կառուցվածքներ (Structures) .....	73
Ցուցիչներ կառուցվածքների վրա (Pointers to Structures) .....	76
Կառուցվածքների խտացում .....	77
Բաժին 3.6 Օգտագործողի հայտարարած տիպեր .....	79
Սեփական տիպերի հայտարարում ( <i>typedef</i> ) .....	79
Միավորումներ (Unions) .....	79
Անանուն (Anonymous) միավորումներ .....	80
Թվարկումներ (Enumerations) ( <i>enum</i> ) .....	81
Բաժին 4.1 Կլասեր (Դասային տիպ) (Classes) .....	83
Կառուցիչներ և փլուզիչներ (Constructors and destructors) .....	85
Կառուցիչների ծանրաբեռնում (Overloading constructors) .....	87
Ցուցիչներ կլասերի վրա (Pointers to classes) .....	89
<i>struct</i> բանալի-բառով որոշված կլասեր .....	90
Բաժին 4.2 Օպերատորների ծանրաբեռնում (Overloading operators) .....	91
Ստատիկ անդամներ (Static members) .....	94
Բաժին 4.3 Կլասերի հարաբերությունները .....	96
Բարեկամ ֆունկցիաներ (Friend functions) .....	96
Բարեկամ կլասեր (Friend classes) .....	97
Կլասերի ժառանգականությունը (Inheritance between classes) .....	98

Ի՞նչ է ժառանգվում ժառանգվող կլասից.....	101
Բազմակի ժառանգականություն (Multiple inheritance) .....	103
Բաժին 4.4 Պոլիմորֆիզմ (Polymorphism).....	104
Ցուցիչներ ժառանգվող կլասերի վրա .....	104
Վիրտուալ անդամներ (Virtual members) .....	105
Աբստրակտ կլասեր (Abstract base classes) .....	106
Բաժին 5.1 Կաղապարներ (templates) .....	109
Ֆունկցիաների կաղապարներ .....	109
Կլասերի կաղապարներ .....	112
Հատուկ կաղապարներ .....	113
Կաղապարի արգումենտներ .....	114
Կաղապարներ և մի քանի ֆայլերով պրոյեկտներ .....	115
Բաժին 5.2 Նախաթարգմանիչի հրամաններ (Preprocessor directives) .....	116
#define .....	116
#undef .....	116
#ifdef, #ifndef, #if, #endif, #else and #elif .....	116
#line .....	117
#error .....	118
#include .....	118
#pragma .....	118
Բաժին 6.1 Մուտք/Ելք ֆայլերի հետ .....	119
Ֆայլի բացում .....	119
Ֆայլի փակում .....	120
Ֆայլերի հետ աշխատանք տեքստային ռեժիմում .....	120
Վիճակի դրոշակների ստուգում .....	121
get և put հոսքային ցուցիչներ .....	122
Երկուական ֆայլեր .....	123
Բուֆերներ և Սինխրոնիզացիա .....	124

## Ներածություն

Այս գիրքը նախատեսված է այն մարդկանց համար, ովքեր ցանկանում են սովորել ծրագրավորել C++ միջավայրում ու անպայման չէ, որ իմանան ինչ-որ այլ ծրագրավորման լեզուներ:

## Թարգմանիչներ

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

Բոլոր C++ի թարգմանիչները կարող են արտադրել Console ծրագրեր: Ստորև բերված են բացատրություններ, թե ինչպես կարելի է արտադրել ծրագրեր Microsoft Visual C++ 6-ի թարգմանիչով:

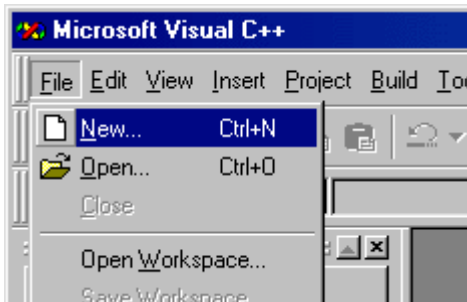
## Console ծրագրերի պրոյեկտներ Microsoft Visual C++ 6 -ով

Այս թարգմանիչը ինտեգրացված է մի ծրագրավորման միջավայրի մեջ, որը կոչվում է Microsoft Visual Studio: Այս ծրագրով ծրագրեր ստեղծելու ամենահեշտ եղանակը՝ պրոյեկտներ ստեղծելն է: Այժմ մենք կստեղծենք **test** անունով պրոյեկտ:

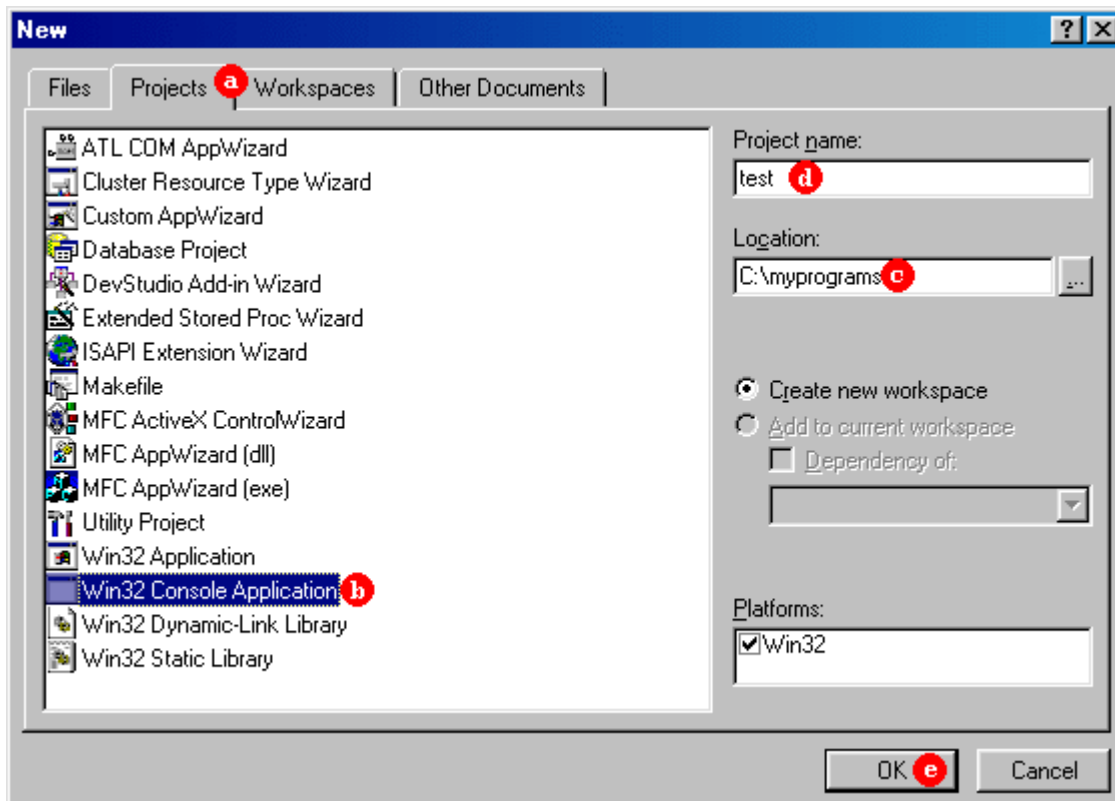
### Նոր պրոյեկտի ստեղծում

1. Աշխատացրեք Microsoft Visual C++ միջավայրը:

Երբ հայտնվի պատուհանը սեղմեք **F**ile այնուհետև **N**ew:



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



Կատարեք հետևյալ քայլերը.

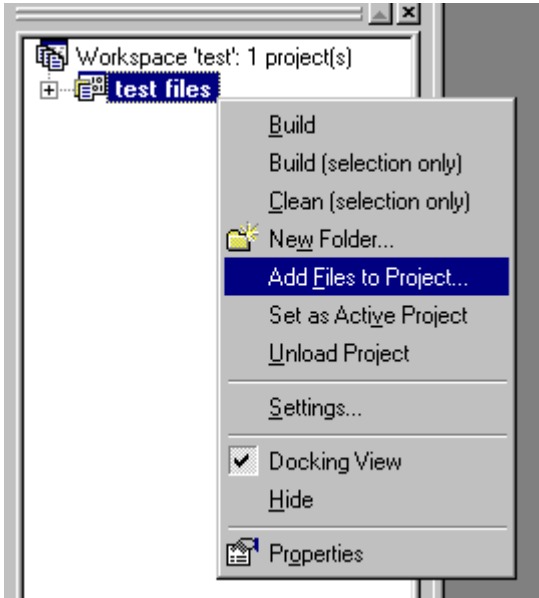
- Ընտրեք **Projects** բաժինը:
- Ընտրեք **Win32 Console Application** պրոյեկտի տիպը:
- Գրեք այն տեղի հասցեն, որտեղ ուզում եք հիշել ձեր ծրագիրը:
- Տվեք ձեր պրոյեկտին անուն, օրինակ՝ **test**:
- Սեղմեք **Ok** կոճակը:

Այնուհետև կհայտնվի մի պատուհան, որը կհարցնի ձեզ, թե ինչպիսի պրոյեկտ եք դուք ուզում ստեղծել: Ընտրեք **An empty project** և սեղմեք **Finish** կոճակը:

3. Այժմ մենք ունենք դատարկ պրոյեկտ: Ձախից ներքևի մասում դուք կտեսնեք երկու բաժիններ՝ **ClassView** և **FileView**: Ընտրեք **FileView** բաժինը:

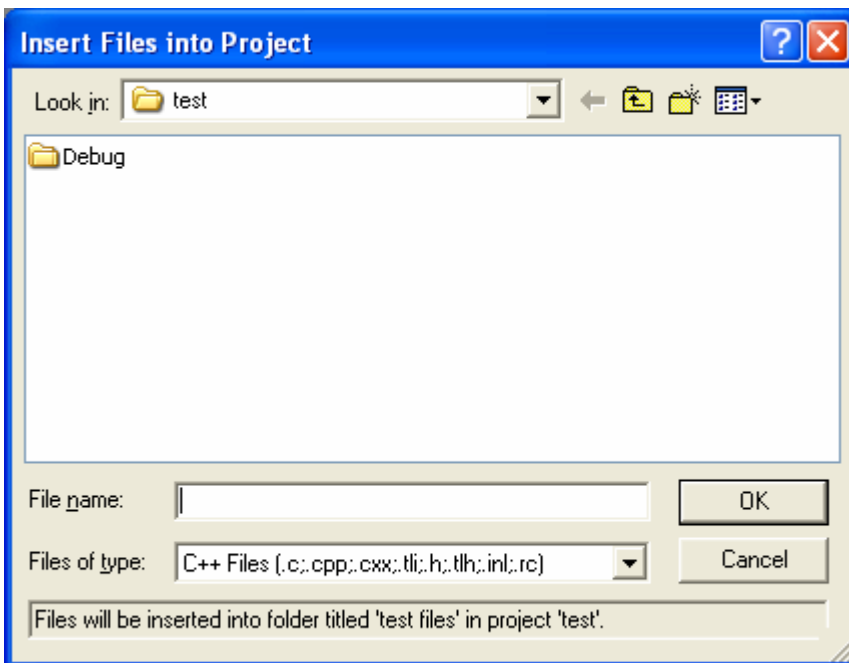


4.

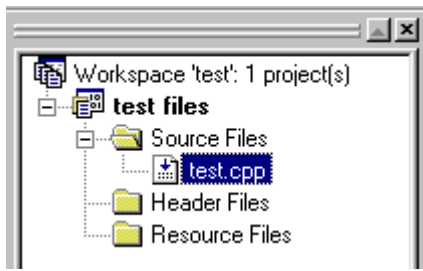


**FileView** բաժնում դուք կտեսնեք **test files** (փոխարինեք **test** բառը այն անունով, որը դուք տվել եք պրոյեկտին **2d** քայլում) անունով մի խումբ: Պահելով մուկը այդ խմբի վրա՝ սեղմեք մկան աջ կոճակը և բերված ցուցակից ընտրեք **Add Files to Project...**:

5. Այս պատուհանի միջոցով մենք կարող ենք ավելացնել ֆայլեր մեր պրոյեկտին: Դուք կարող եք ավելացնել կամ արդեն գոյություն ունեցող ֆայլ, կամ էլ ստեղծել նորը, գրելով նոր ֆայլի անունը **File name** տեքստային պատուհանի մեջ: Ձեր ստեղծած ֆայլի վերջավորությունը կլինի **cpp**, ինչը նշանակում է **C Plus Plus**: Նոր ֆայլ ստեղծելուց ծրագիրը կհարցնի ձեզ, թե արդյոք դուք ցանկանում եք ստեղծել նոր ֆայլ: Պատասխանեք այո՝ Yes:




6. Այն բանից հետո, երբ դուք կավելացնեք ֆայլեր ձեր պրոյեկտին, նրանք կհայտնվեն **test files** խմբի տակ՝ **Source Files** կատալոգում: Մկան ձախ կոճակով՝ կրկնակի անգամ սեղմելով ֆայլի վրա, ֆայլի պարունակությունը կերևա մի նոր բացված պատուհանի մեջ և դուք կկարողանաք աշխատել նրա հետ:



## Պրոյեկտի կոմպիլացիա (compilation) և աշխատեցում (execution)

Եթե ծրագիրը արդեն պատրաստ է և դուք ուզում եք աշխատողեցնել այն, ապա ընտրեք **Build** մենյուն և բացված ցուցակից ընտրեք **Execute test.exe** (test.exe բառը կփոխվի, կախված ձեր պրոյեկտի անունից) տողը:

Այս գործողությունը կարող է կատարվել նաև  սիմվոլը սեղմելով:

Եթե ձեր ծրագրում սխալներ չկան, ապա այն աշխատացնելուց հետո դուք կտեսնեք ծրագրի արդյունքը, հակառակ դեպքում, էկրանի ներքևի մասում, կտեսնեք չաշխատելու պատճառը:



## Բաժին 1.1

### C++ի կառուցվածքը

Ծրագրավորման լեզու սովորելու ամենալավ եղանակը այն օրինակների վրա սովորելն է: Ստորև բերված է առաջին ծրագիրը.

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

Վերևի մասում գրված է մեր առաջին ծրագիրը, որը մենք կարող ենք անվանել, օրինակ՝ barev.cpp: Ներքևի մասում բերված է ծրագրի արդյունքը աշխատեցնելուց հետո: Սա այն հեշտագույն 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 մեկնաբանություն 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 մեր ծրագրի խորագիր:

#include <iostream.h>

Այն նախադասությունները, որոնք սկսվում են # նշանով նախապրոցեսորի հրամաններ են (preprocessor directives), որոնք օգտագործվում են թարգմանիչին հրամաններ տալու համար: Մեր դեպքում #include <iostream.h> տողը ասում է թարգմանիչի նախապրոցեսորին ներառել iostream անունով ստանդարտ գրադարանը կամ, ինչ նույնն է, header ֆայլը: Այս ֆայլի մեջ նկարագրված է C++ի ստանդարտ մուտք-ելքի գրադարանը, որը օգտագործվում է ծրագրի հաջորդ տողերում:

int main ()

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

**main** բառին անմիջապես հաջորդում են փակագծեր ( ), քանի որ սա ֆունկցիա է: C++-ում բոլոր ֆունկցիաներին անմիջապես հաջորդում են փակագծեր ( ), որոնք, անհրաժեշտության դեպքում, կարող են պարունակել արգումենտներ: **main**-ում գրված ծրագիրը անմիջապես հաջորդում է ֆունկցիայի հայտարարությանը և 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), և այս տողով մենք ելքի հոսքին ուղղարկում ենք սիմվոլների հաջորդականություն (մեր դեպքում՝ “Barev!”): **cout**-ը հայտարարված է **iostream.h** header ֆայլում, և որպեսզի կարողանալ օգտագործել այն, հարկավոր է “հրամայել” թարգմանիչին ներառել այդ ֆայլը:

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

```
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 բոլոր C++-ով գրված ծրագրերը ավարտվում են այս տողին նման տողով:

Սակայն, երևի թե դուք նկատեցիք, որ այս ծրագրի ոչ բոլոր տողերն են կատարում ինչ-որ գործողություն: Առաջին երկու տողերը պարունակում էին միայն մեկնաբանություն (սրանք սկսվում են // սիմվոլներով) և նախապրոցեսային հրաման (սրանք սկսվում են # սիմվոլով), հաջորդ տողում հայտարարվում էր **main( )** ֆունկցիան և վերջապես վերջին տողերում գրված էր **cout**-ին դիմելու հրամանը, սահմանափակված ձևավոր փակագծերով:

Ծրագիրը գրված է մի քանի տողով, այն կարդալը և հասկանալը հեշտացնելու նպատակով, սակայն նույն ծրագիրը կարող ենք գրել հետևյալ կերպ՝

```
#include <iostream.h>
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.h>

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

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

Այս դեպքում մենք օգտագործեցինք `cout <<` մեթոդը (method) երկու անգամ՝ իրարից տարբեր երկու հրամանների մեջ: Մեկ անգամ ևս նշենք, որ ծրագրի բաժանումը մի քանի տողերի կատարված է միայն կարդալու հարմարավետության նպատակով և **main( )** ֆունկցիան մենք 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 նախապրոցեսորի հրամանները (որոնք սկսվում են # սիմվոլով) չեն ենթարկվում այս կանոնին, քանի որ նրանք ծրագրի հրամաններ չեն: Նրանք տողեր են, որոնք կարդացվում և փոխարինվում են նախապրոցեսորի կողմից և ոչ մի ծրագիր չեն արտադրում: Նրանք պետք է գրված լինեն նոր տողի վրա և չեն պահանջում ; սիմվոլը տողի ավարտին:

## Մեկնաբանություններ (Comments)

Մեկնաբանությունները ծրագրի կտորներ են, որոնք անտեսվում են թարգմանիչի կողմից: Նրանք ոչինչ չեն անում: Նրանց իմաստը միայն այն է, որ ծրագրավորողը, ցանկության դեպքում, ծրագրի մեջ կարողանա գրել հուշումներ և բացատրություններ:

C++-ում կա մեկնաբանություն հայտարարելու երկու ձև՝

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

Սրանցից առաջինը՝ տողային մեկնաբանություն, անտեսում է ամեն ինչ տրված տողում, սկսած // սիմվոլներից: Երկրորդը՝ բլոկային մեկնաբանություն, անտեսում է /\* \*/ սիմվոլների միջև ամեն ինչ:

Ավելացնենք որոշ մեկնաբանություններ մեր երկրորդ ծրագրին.

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

#include <iostream.h>

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 դուք գրեք ձեր մեկնաբանությունները ծրագրի մեջ առանց օգտագործելու վերը նշված ձևերից որևէ մեկը, ապա թարգմանիչը կընդունի ձեր գրվածը, որպես տարբեր հրամաններ և հավանաբար կտա սխալ:

## Բաժին 1.2

### Փոփոխականներ: Ինֆորմացիայի 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 ծրագիրը, որպեսզի ստանալ մի նախադասություն էկրանի վրա: Շատ ավելի հեշտ և արագ կլինեք ինքնուրույն գրել տրված նախադասությունը, բայց 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 ծանոթանալ **փոփոխականի** (variable) գաղափարի հետ:

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 առաջին ասած թիվը ավելացնել 1-ով, դուք պետք է մտքում ունենաք  $6$  ( $5 + 1$ ) և 2 թվերը: Հիմա կարող ենք այս թվերը իրարից հանել և ստանալ 4:

Այն գործողությունները, որոնք դուք կատարեցիք շատ նման է նրան, թե 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, բայց ձեր համակարգիչը կարող է միաժամանակ պահել միլիոնավոր այդպիսի թվեր և դրանց հետ կատարել բարդ մաթեմատիկական գործողություններ:

Այսպիսով մենք կարող ենք հայտարարել 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:

## Նույնարկիչներ (Identifiers)

Վավեր նույնարկիչը 1 կամ ավել 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`-եր: Բացի դրանից փոփոխականի նույնարկիչը չի կարող սկսվել թվով:

Նույնարկիչը նաև չի կարող համընկնել C++ լեզվում սահմանված **բանալի-բառերի** (Key Word) հետ: Հետևյալ 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 մեծատառերը և փոքրատառերը տարբեր ձևի են հասկացվում. այսինքն՝ օրինակ RESULT և result նույնարկիչները միմյանցից տարբեր են:

## Ինֆորմացիայի տիպեր (Data types)

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

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

Ինֆորմացիայի տիպեր

Անուն	Բայթեր*	Բացատրություն	Միջակայք*
<b>char</b>	1	մեկ սիմվոլ կամ 8 բիթ երկարությամբ ամբողջ թիվ	<b>signed:</b> -128 to 127 <b>unsigned:</b> 0 to 255
<b>short</b>	2	16 բիթ երկարությամբ ամբողջ թիվ	<b>signed:</b> -32768 to 32767 <b>unsigned:</b> 0 to 65535
<b>long</b>	4	32 բիթ երկարությամբ ամբողջ թիվ	<b>signed:</b> -2147483648 to 2147483647 <b>unsigned:</b> 0 to 4294967295
<b>int</b>	*	Ամբողջ թիվ: Երկարությունը կախված է այն օպերացիոն համակարգից, որի վրա աշխատում ենք: MSDOS համակարգերի վրա երկարությունը 16 բիթ է, իսկ Windows 9x/2000/NT և նմանատիպ համակարգերի դեպքում՝ 32 բիթ:	Տե՛ս <b>short</b> -ի, <b>long</b> -ի միջակայքերը

<b>float</b>	4	Լողացող կետիկով թվեր: Կոտորակային թվերն են՝ 124.46878654	3.4e + / - 38 (7 նիշ)
<b>double</b>	8	Կրկնակի ճշտության լողացող կետիկով թվեր	1.7e + / - 308 (15 նիշ)
<b>long double</b>	10	Կրկնակի ճշտության լողացող կետիկով երկար թվեր	1.2e + / - 4932 (19 նիշ)
<b>bool</b>	1	Բուլյան արժեքներ: Կարող է ընդունել միայն երկու հնարավոր արժեք՝ true (ճիշտ) կամ false (ոչ ճիշտ)	true կամ false
<b>wchar_t</b>	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 կան թարգմանիչներ, որոնք այս տիպը դեռևս չունեն	լայն սիմվոլ

Բացի այս տիպերից կան նաև **ցուցիչներ** (pointers) և **դատարկ** (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. սկզբից գրում ենք տեսակը, ինչպիսին ուզում ենք լինի փոփոխականը (օրինակ՝ int, short, float ...) այնուհետև վավեր փոփոխականի նույնարկիչ: Օրինակ՝

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

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

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

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

հայտարարում է int տեսակի 3 փոփոխականներ (a, b և c) և ունի ճիշտ նույն նշանակությունը, ինչ եթե գրեինք.

```
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 տիպից առաջ գրում ենք signed կամ unsigned: Օրինակ՝

```
unsigned short MardkancQanak;  
signed int AmsekanHashvekshir;
```

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

```
int AmsekanHashvekshir;
```

որը կունենար ճիշտ նույն նշանակությունը:

Որպեսզի ցուցադրել, 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.h>  
  
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

Մի՛ անհանգստացեք, եթե որոշ բաներ մի-քիչ տարօրինակ են թվում: Դուք ամեն ինչին ավելի մանրամասն կծանոթանաք հաջորդ գլուխներում:

## Փոփոխականների սկզբնարժեքավորում (Initialization of variables)

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

*տիպ նույնարկիչ = արժեք ;*

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

```
int a = 0;
```

Բացի սրանից կա նաև մեկ այլ տարբերակ՝

*տիպ նույնարկիչ (արժեք) ;*

Օրինակ՝

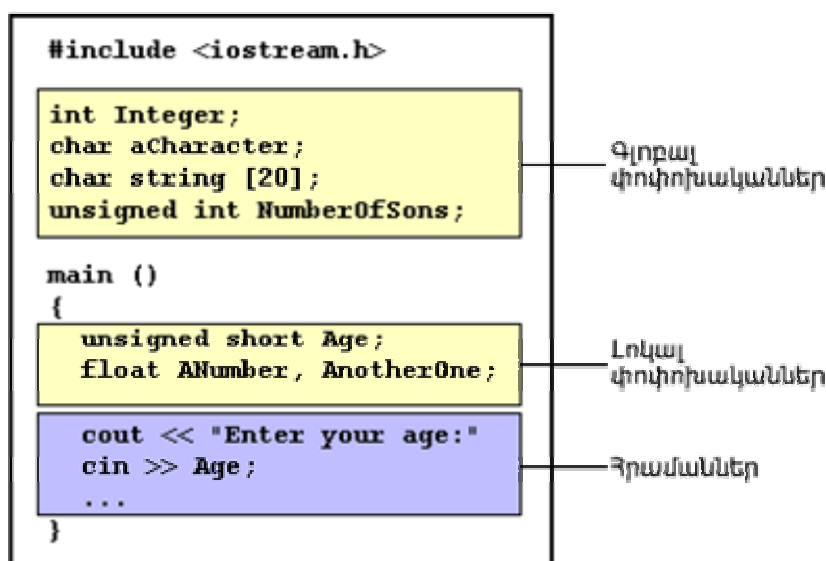
```
int a ( 0 );
```

C++ լեզվում այս երկուսը ճիշտ նույն նշանակությունը ունեն:

## Փոփոխականների տեսանելիության տիրույթ (Scope of variables)

Բոլոր փոփոխականները, որոնք ուզում ենք օգտագործել, պետք է նախապես հայտարարված լինեն: C++ լեզվում փոփոխականները կարելի է հայտարարել կոդի մեջ ցանկացած տեղ:

Սակայն խորհուրդ է տրվում փոփոխականների հայտարարությունը անջատել կոդի մնացած մասից:



**Գլոբալ փոփոխականներին** (Global variables) կարելի է դիմել կոդի մեջ ցանկացած տեղ՝ իր հայտարարությունից հետո:

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



## 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 թարգմանիչին հասկացնենք, որ թիվը

8-ական է, նրանից առաջ գրում ենք 0, իսկ 16-ականներից առաջ՝ 0x: Օրինակ հետևյալ 3 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  $\pi$ -ն է, երկրորդը՝ Ավոգադրիոյի հաստատունը, հաջորդը՝ էլեկտրոնի լիցքն է, վերջինը՝ բնական 3 թիվը՝ արտահայտած որպես լողացող կետով թիվ:

### Սիմվոլներ և տողեր (Characters and strings)

Կան նաև ոչ թվային հաստատուններ՝

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

Առաջին 2 արտահայտությունները առանձին սիմվոլներ են, իսկ մյուս երկուսը իրենցից ներկայացնում են տողեր: Նկատենք, որ միայնակ սիմվոլը մենք վերցնում ենք (') սիմվոլների միջև իսկ տողերը (") սիմվոլների միջև:

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

\n	նոր տող
\r	Enter կոճակի կողք
\t	Տաբուլացիա
\v	Ուղղահայաս տաբուլացիա
\b	Ջնջել մեկ սիմվոլ
\f	Նոր էջ
\a	Զանգ
\'	' սիմվոլը
\"	" սիմվոլը
\?	? սիմվոլը
\\	\ 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 1-ից ավել 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 դրանց 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, դրանից հետո հանդիպող *նույնարկիչ* բառը փոխարինում է *արժեք* արտահայտությամբ:

## Հայտարարված հաստատուններ (Declared constants) (*const*)

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

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

## Բաժին 1.3

### Օպերատորներ

Այժմ, երբ մենք արդեն գիտենք փոփոխականների և հաստատունների գոյության մասին, մենք կարող ենք սկսել աշխատել նրանց հետ: Այդ նպատակով C++-ը տրամադրում է մեզ օպերատորներ, որոնք իրենցից ներկայացնում են նշանների և յուրահատկիչների ցուցակ, որոնք այբուբենի տառեր չեն, բայց ներկա են բոլոր ստեղծագործությունների վրա: Շատ կարևոր է իմանալ սրանք, քանի որ այս օպերատորները կազմում են C++-ի անբաժանելի մասը:

Հարկ չէ անգիր անել այս թեման ամբողջությամբ, մանրամասն նկարագրությունը բերված է հետագայի համար ուղղեցույց ծառայելու նպատակով:

#### Վերագրում (=)

Վերագրման օպերատորը ծառայում է փոփոխականին արժեք վերագրելու համար:

```
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, սրանց ինչ-որ օպերացիայի կամ կոմբինացիայի արդյունք:

Հարկ է նշել, որ վերագրման գործողությունը միշտ տեղի է ունենում աջից ձախ, այլ ոչ հակառակը:

```
a = b;
```

վերագրում է **a** փոփոխականին **b**-ի արժեքը, նկատի չունենալով տվյալ պահին **a**-ում պահվող արժեքը: Հաշվի առեք նաև այն հանգամանքը, որ մեքն միայն **a**-ին վերագրում ենք **b**-ի արժեքը և **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 մենք գրել էինք՝ **a = b**;

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

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

համարժեք է հետևյալին՝

```
b = 5;
a = 2 + b;
```

ինչ 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;
```

որը վերագրում է **5** թիվը 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 օգտագործելը առանձնահատուկ հասկացման կարիք չեն պահանջում, քանի որ տարրական մաթեմատիկայից դուք արդեն գիտեք նրանց օգտագործման ձևը:

Միայն **մոդուլ**-ը կարող է ձեզ համար անհասկանալի լինի: **մոդուլ**-ը այն օպերացիան է, որը վերադարձնում է երկու ամբողջ թվերի բաժանման հետևանքից ստացված մնացորդը: Օրինակ եթե մենք գրենք **a = 11 % 3**;, ապա **a**-ի արժեքը կլինի **2**, քանի որ եթե մենք բաժանենք **11**-ը **3**-ի վրա, կստանանք **2** մնացորդ:

### Բարդ վերագրման օպերատորներ (+=, -=, \*=, /=, %=, >>=, <<=, &=, ^=, |=)

Այս օպերատորների միջոցով հնարավորություն է տրվում ծրագրավորողին փոփոխել 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);
```

և նույնը բոլոր մնացած օպերատորների համար:

### Մեծացման և փոքրացման օպերատորներ.

Մյուս օպերատորները, որոնց միջոցով մենք կարող ենք փոփոխել 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;
```

բոլոր արտահայտությունները 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++** կամ **++a** պարզ արտահայտությունները ունեն միևնույն նշանակությունը, բայց ուրիշ դեպքերում, որտեղ *մեծացման* և *փոքրացման* գործողությունը հաշվում է, որպես առանձին (ուրիշ) արտահայտություն, նրանք կարող են ունենալ բացարձակապես տարբեր նշանակություն. այն դեպքում, երբ մեծացման օպերատորը օգտագործվում է, որպես prefix (**++a**), **a**-ի արժեքը մեծացվում է, մինչև արտահայտության արժեքի հաշվումը և արդեն այդ հաշվման մեջ մասնակցում է նոր (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;
```

Այս դեպքում **A**-ն **4** է, **B**-ն՝ **4** է:

Օրինակ 2.

```
B=3;  
A=B++;
```

Այս դեպքում **A**-ն **3** է, **B**-ն՝ **4** է

#### Համեմատության օպերատորներ (==, !=, >, <, >=, <=)

Երկու արտահայտություններ իրար հետ համեմատելու համար մենք կարող ենք օգտագործել համեմատության օպերատորները: Համեմատության գործողության արդյունքը լինելու է **bool** տիպի, որը կընդունի **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  $2*3 >= 6$ :

(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, իսկ երկրորդը՝ համեմատության օպերատոր է: Այդ է պատճառը, որ վերջին՝ ((b=2) == a) օրինակում մենք սկզբից **b**-ին վերագրեցին **2**՝ (b=2), իսկ հետո նոր համեմատեցինք **b**-ն **a**-ի հետ: Քանի որ **a**-ն **2** էր, համեմատման արդյունքը ստացվեց **true**:

### Տրամաբանական օպերատորներ (!, &&, ||)

! օպերատորը համարժեք է բուլյան ժխտման օպերատորին և նա պահանջում է միայն մեկ օպերանդ (operand), գրված նրանից աջ: Փաստորեն այս օպերատորի իմաստը արժեքի շրջումն (ժխտումն) է. վերադարձնել **false**, եթե օպերանդը **true** է և **true**, եթե օպերանդը **false** է: Սա նույնն է, ինչ 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 օպերատորները օգտագործվում են, երբ գնահատվում են երկու արտահայտություններ՝ մեկ պատասխան ստանալու նպատակով: Սրանք համապատասխանում են համապատասխանաբար բուլյան *կոնյունկցիա* և բուլյան *դիզյունկցիա* ֆունկցիաներին: Գործողության արդյունքը կախված է երկու օպերանդներից.

Առաջին օպերանդ <b>a</b>	Երկրորդ օպերանդ <b>b</b>	Արդյունք <b>a &amp;&amp; b</b>	Արդյունք <b>a    b</b>
true	true	<b>true</b>	<b>true</b>
true	false	<b>false</b>	<b>true</b>
false	true	<b>false</b>	<b>true</b>

false	false	<b>false</b>	<b>false</b>
-------	-------	--------------	--------------

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 *պայմանը* **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
&	<b>AND</b>	Տրամաբանական ԵՎ
	<b>OR</b>	Տրամաբանական ԿԱՄ
^	<b>XOR</b>	Տրամաբանական Բացառիկ ԿԱՄ
~	<b>NOT</b>	Լրացում
<<	<b>SHL</b>	Ձախ տեղաշարժ
>>	<b>SHR</b>	Աջ տեղաշարժ

### Տիպերի ձևափոխման օպերատորներ

Տիպերի ձևափոխման օպերատորները թույլ են տալիս մի տիպը վերածել մեկ այլ տիպի: C++-ում կան մի քանի եղանակներ այդ բանը 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 վերածում է **3.14 float** տիպի թիվը ամբողջ՝ **int** տիպի: Այստեղ, որպես տիպի ձևափոխման օպերատոր, հանդիսանում է (**int**)-ը: Այս նույն բանը կատարելու մյուս



եղանակը՝ կառուցիչի (constructor) օգտագործումն է. գրել նոր տիպը և, այնուհետև, փակագծերի մեջ գրել ձևափոխվող արտահայտությունը.

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

Ներկայացված երկու ձևափոխման եղանակներից բացի կան նաև այլ եղանակներ, որոնք կներկայացվեն հաջորդ դասերում:

### sizeof()

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

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

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

**sizeof**-ի կողմից վերադարձվող թիվը հաստատուն է, այսինքն այն արդեն որոշված է ծրագրի աշխատանքից առաջ:

## Օպերատորների նախապատվությունը

Բարդ արտահայտություններ կազմելուց, մեր մոտ կարող է առաջանալ հարց, թե որ գործողությունը կկատարվի ավելի շուտ: Օրինակ  $a = 5 + 7 \% 2$  արտահայտության մեջ կարող է կասկած առաջանալ, թե արդյոք այս արտահայտությունը կհամարվի  $a = 5 + (7 \% 2)$  արտահայտությունը **6** արդյունքով, թե՛  $a = (5 + 7) \% 2$ , **0** արդյունքով: Ճիշտ պատասխանը առաջին դեպքն է՝ **6**: C++-ում գոյություն ունի օպերատորների (ոչ միայն թվաբանական, այլ բոլոր օպերատորների) նախապատվությունների հաջորդականություն, ըստ որի այն որոշում է, թե որ գործողությունը պետք է կատարվի ավելի շուտ: Ստորև բերված է այդ ցանկը.

Նախ.	Օպերատոր	Բացատրություն	Ասոցիատիվություն
1	::	Տեսանելիության տիրույթ	ձախ
2	() [ ] -> . sizeof		ձախ
3	++ --	մեծացում/փոքրացում	աջ
	~	բիթային լրացում (bitwise)	
	!	ունար ՈՉ	
	& *	հասցեավորում և ետհասցեավորում	
	(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

*Ասոցիատիվությունը* 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 հարմարավետ, փակագծերի միջոցով: Օրինակ.

$a = 5 + 7 \% 2;$

խորհուրդ է տրվում գրել

$a = 5 + (7 \% 2);$  կամ

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

կախված այն բանից, թե որ գործողությունն ենք ուզում կատարվի ավելի շուտ:

Այդ պատճառով, եթե դուք ուզում եք գրել բարդ արտահայտություն և դուք համոզված չեք, թե որ գործողությունն է կատարվում 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 սարքը:

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 դուք կարող եք աշխատել օգտագործողի հետ՝ նրան ցույց տալ ինֆորմացիա և նրանից ստանալ հրամաններ:

### Ելք (Output) (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 փոփոխականը **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";              // ekrani vra tpum e barev
cout << barev;                 // ekrani vra tpum e barev popoxalani parunakutyun@
```

Տեղադրման օպերատորը (<<) մեկ նախադասության 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 նախադասության 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 դարձնենք, 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 սիմվոլ (**\n**)՝

```
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;
```

կտալի՝

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

## Մուտք (Input) (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 ինֆորմացիա մուտքագրվի. ստացված ինֆորմացիան պահվում է **tariq** փոփոխականի մեջ:

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

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

```
// mutqi-yelqi orinak
#include <iostream.h>

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.**

Ծրագիրն օգտագործողը ծրագրում սխալներ առաջանալու հիմնական պատճառն 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 ինֆորմացիա տա (օրինակ՝ դու հարցնես տարիքը և սպասես, որ նա վերադարձնի ամբողջ տեսակի թիվ, իսկ նա գրի իր անունը՝ սխմվոլների տող): Հիմա դեռ ստիպված կլինենք լիովին վստահել օգտագործողին, սակայն, շարունակելով ուսումնասիրել այս ձեռնարկը, շուտով կգտնենք այս պրոբլեմի որոշակի լուծումներ:

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

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

**համարժեք է ինչ՝**

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

## Բաժին 2.1

### Կառավարման ստրուկտուրաներ (համակարգեր) (Control structures)

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

Կառավարման համակարգերի մեծ մասը, որոնք մենք այժմ կանցնենք, 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 կցանկանաք գրել մի քանի հրամաններ, ապա պարտադիր այդ հրամանները պետք է սահմանափակված լինեն ձևավոր փակագծերով՝ առաջացնելով հրամանների բլոկ:

### Պայմանային ստրուկտուրա (համակարգ) *if* և *else* (Conditional structure: *if* and *else*)

Այս համակարգի տեսքը հետևյալն է.

```
if (պայման) մարմին
```

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

Օրինակ ստորև բերված ծրագիրը կտպի **x-ը 100 է** այն և միայն այն դեպքում, երբ **x** -ը իրոք լինի 100.

```
if (x == 100)
    cout << "x-ը 100 է";
```

Եթե մենք ցանկանանք, որ կատարվի մեկից ավելի հրաման *պայմանը true* լինելու դեպքում, ապա պետք է գրենք այդ հրամանները ձևավոր փակագծերի մեջ.

```
if (x == 100)
{
    cout << "x-ը ";
    cout << x << " է";
}
```

Մենք նաև կարող ենք սահմանել գործողություն այն դեպքի համար, երբ *պայմանը true* չէ՝ օգտագործելով **else** բանալի-բառը: Նա օգտագործվում է միայն **if**-ի հետ միասին 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* ստրուկտուրաների միջոցով, մենք 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, եթե դուք ուզում եք օգտագործել մեկից ավելի հրաման, 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 ստրուկտուրաներ (համակարգեր) կամ ցիկլեր (Repetitive structures or loops)

*Ցիկլերի* նպատակն է կրկնել *մարմինը* մի քանի անգամ, կամ քանի դեռ պայմանը ճիշտ է:

### **while** ցիկլը

Տեսքը.

**while** (*պայման* ) *մարմին*

և նրա ֆունկցիան միայն *մարմինը* կրկնելն է, քանի դեռ *պայմանը* ճիշտ է:

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

```
#include <iostream.h>
int main ()
{
    int n;
```

```

cout << "Nermucec 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 աշխատացվում է, օգտագործողը խնդրվում է ներմուծել մի 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 վերագրում է արժեք  $n$ -ին:
- 2. *while* հրամանը ստուգում է, թե արդյոք ( $n > 0$ ). Այստեղ հնարավոր է երկու դեպք.
  - **true**. կատարվում է *մարմինը* (քայլ 3),
  - **false**. բաց է թողնվում մարմինը և կատարվում է 5-րդ քայլը:
- 3. Կատարվում է *մարմինը*.  
`cout << n << ", ";`  
`--n;`  
 (տպում է  $n$ -ը էկրանին, իսկ հետո պակասեցնում այն 1-ով):
- 4. Բլոկը ավարտվում է: Վերադարձ 2-րդ քայլին:
- 5. Կատարվում է բլոկից հետո եկող ծրագիրը՝ էկրանին տպվում է **VERJ!** և ծրագիրը ավարտվում է:

Նշենք նաև այն հանգամանքը, որ ցիկլը պիտի ունենա ավարտ, այսինքն, ցիկլի *մարմնում* պետք է ներկա լինի, *պայմանը false* դարձնող, մի ինչ-որ արտահայտություն, որն էլ և կստիպի ցիկլին ավարտվել: Մեր դեպքում, որպես այդպիսի արտահայտություն, հանդես է գալիս `--n`; տողը, որը ցիկլի որոշակի կրկնություններից հետո  $n$ -ը 0 է դարձնում, որն էլ և ստիպում է ցիկլին ավարտվել:

## do-while ցիկլը

Տեսքը.

```
do մարմին while (պայման);
```

Այս ցիկլը լրիվ նույնն է, ինչ *while* ցիկլը, միայն այն տարբերությամբ, որ *պայմանը* ստուգվում է ոչ թե ցիկլի սկզբում, այլ վերջում՝ դրանով ապահովվելով *մարմնի*՝ գոնե մեկ անգամվա կատարումը:

Օրինակ հետևյալ ծրագիրը կխնդրի օգտագործողին մուտքագրել ինչ-որ մի թիվ այնքան անգամ, քանի դեռ մուտքագրված թիվը 0 չի լինի: 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*;

Սրա հիմնական ֆունկցիան՝ *մարմինը* կրկնելն է, քանի դեռ *պայմանը* ճիշտ է, ինչպես և *while* - ում։ Բայց այս ցիկլը նաև թույլ է տալիս սահմանել *սկզբնարժեքավորման* և *փոփոխություն* օպերատորները։

Ահա, թե ինչպես է այն աշխատում.

1. կատարվում է *սկզբնարժեքավորման* 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 փակագծերի մեջ։
4. վերջապես, կատարվում է *փոփոխություն* օպերատորը և ցիկլը վերադառնում է 2-րդ քայլին։

Օրինակ.

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

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

Սկզբնաբժեքավորման և փոփոխության օպերատորները պարտադիր չեն: Նրանք կարող են բաց թողնվել, բայց ոչ կետ-ստորակետերը: Օրինակ, մենք կարող ենք գրել` `for (;n<10;)`, եթե մենք չենք ուզում իրականացնել որևէ սկզբնաբժեքավորում և փոփոխություն, կամ` `for (;n<10;n++)`, եթե ուզում ենք սահմանել փոփոխության օպերատորը, բայց չենք ուզում իրականացնել որևէ սկզբնաբժեքավորում:

Օգտագործելով ստորակետ (,) օպերատորը, մենք կարող ենք գրել մի քանի հրամաններ `for` –ի օպերատորներից յուրաքանչյուրում` օրինակ *սկզբնաբժեքավորման* մեջ: Ստորակետ (,) օպերատորը` հրամանների տարանջատիչ է և նա 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, i=100 ; n!=i ; n++, i-- )
{
    // մեր ծրագիրը
}
```

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

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

Սկզբնաբժեքավորում      Պայման      Փոփոխություն

`n`-ը սկզբնաբժեքավորվում է 0, իսկ `i` –ն` 100, և *պայմանը* (`n!=i`) (`n`-ը հավասար չէ `i`-ի) է: Քանի որ `n`-ը մեծացվում է մեկով, իսկ `i`-ն փոքրացվում է մեկով, ապա ցիկլի *պայմանը* կդառնա `false` 50 քայլ հետո, երբ `n`-ը և `i`-ն հավասար կլինեն 50:

## Անցման հրամաններ (Jumps)

### *break* հրամանը

Օգտագործելով *break* հրամանը, մենք կարող ենք լքել ցիկլը` նույնիսկ, եթե դրա ավարտի պայմանը չի կատարվել: Այս հրամանը կարող է օգտագործվել անվերջ ցիկլներից դուրս գալու համար, կամ հարկադրական եղանակով ցիկլից դուրս գալու նպատակով: Օրինակ հետևյալ ծրագրում ցիկլը պետք է ավարտվի 3 թվին հասնելիս.

```
#include <iostream.h>
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, իբրև հասել է ցիկլի ավարտին, ինչը և հանգեցնում է ցիկլի կրկնմանը: Հետևյալ օրինակում մենք շրջանցելու ենք 5 թիվը:

```

#include <iostream.h>
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 ծառայում է նաև որպես *goto* հրամանի արգումենտ: Նշիչը իրենից ներկայացնում է իդենտիֆիկատոր, որին հաջորդում է (:) սիմվոլը:

Օրինակ.

```

#include <iostream.h>
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!

## **Ընտրության 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:
        լռությամբ հրամանների բլոկ
}
```

Սա աշխատում է հետևյալ կերպ. սկզբից *switch*-ը գնահատում է *արտահայտությունը* և ստուգում է, թե արդյոք այն էկվիվալենտ է *հաստատուն1*-ին: Եթե դա իրոք այդպես է, ապա կատարվում է *հրամանների բլոկ 1* –ը, մինչև **break**-ին հանդիպելը, որից հետո «թոնում է» *switch*-ի վերջին: Եթե *արտահայտությունը* էկվիվալենտ չէ *հաստատուն1*-ին, ապա ստուգվում է *արտահայտության* էկվիվալենտությունը *հաստատուն2*-ի հետ: Եթե դա իրոք այդպես է, 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 որ կցանկանաք), ապա կկատարվի *լռությամբ հրամանների բլոկը*, եթե այն ներկա է:

Ստորև բերված երկու ծրագրերն իրար համարժեք են.

#### **switch-ի օրինակ**

```
switch (x) {
    case 1:
        cout << "x is 1";
        break;
    case 2:
        cout << "x is 2";
        break;
    default:
        cout << "value of x unknown";
}
```

#### **if-else-ով էկվիվալենտ օրինակ**

```
if (x == 1) {
    cout << "x is 1";
} else {
    if (x == 2) {
        cout << "x is 2";
    } else {
        cout << "value of x unknown";
    }
}
```

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 չէ սահմանափակել 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* ուրիշ հաստատունների հետ համեմատելու համար: Օրինակ դուք չեք կարող, որպես *հաստատուն* գրել փոփոխականներ (**case (n\*2):**): Եթե դուք ուզում եք համեմատել տրված *արտահայտությունը* փոփոխականների կամ ինչ-որ տիրույթների հետ, ապա օգտագործեք **if-else** հրամանները:

## Բաժին 2.2

### Ֆունկցիաներ ( Functions ) (I)

Օգտագործելով ֆունկցիաները մենք կարող ենք ծրագիրը բաժանել առանձին հատվածների՝ հնարավորություն ստանալով արդեն պատրաստ ծրագրի մասերը 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, արգումենտ 2, ... ) մարմին**

որտեղ.

- **տիպը** ֆունկցիայի վերադարձվող արժեքի տիպն է,
- **անունը** այն անունն է, որը պետք է օգտագործենք ֆունկցիան կանչելիս,
- **արգումենտները** (կարելի է ունենալ այնքան արգումենտ, որքան անհրաժեշտ է): Ամեն արգումենտ կազմված է տիպից և նույնարկիչից, որով այն պիտի տարբերվի մյուսներից (օրինակ՝ int x). սա նման է փոփոխականների հայտարարման, և նշված արգումենտը ֆունկցիայի մեջ իրեն «պահում է» ճիշտ այնպես, ինչպես ցանկացած փոփոխական: Դրանք հնարավորություն են տալիս ֆունկցիային արգումենտներ փոխանցել: 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.h>

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

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

**Ardayunq@ 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 ես **main** ֆունկցիան սկսվում է **int** տեսակի **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 );
```

Արգումենտները ամբողջովին համապատասխանում են: **main** ֆունկցիայի միջից մենք 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: Ֆունկցիային կանչի ժամանակ 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 տալիս **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** (**return (r);**), որը այդ ժամանակ իր մեջ պահում էր **8** արժեքը. այսպիսով ֆունկցիան վերադարձնում է այդ արժեքը:

```
int gumarum (int a, int b)
↓$
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 << "Ardyunq@ exav " << z;
```

## Փոփոխականների տեսանելիության տիրույթ (Scope of variables) [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: Նաև **gumarum** ֆունկցիայից չենք կարող դիմել **main** ում հայտարարված **z** փոփոխականին:

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

Ֆունկցիաների վերաբերյալ մեկ այլ օրինակ.

```
// funkciayi orinak
#include <iostream.h>

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 ardyunq@: " << z << '\n';
    cout << "Yerkrord ardyunq@: " << hanum (7,2) << '\n';
    cout << "Yerord ardyunq@: " << hanum (x,y) << '\n';
    z= 4 + hanum (x,y);
    cout << "Chorord ardyunq@: " << z << '\n';
    return 0;
}
```

```
Arajin ardyunq@: 5
Yerkrord ardyunq@: 5
Yerord ardyunq@: 2
Chorord ardyunq@: 6
```

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

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

Քննարկենք օրինակը՝ տող առ տող.

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

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

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



Նույնը հաջորդ տողի մասին.

```
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 կանչվելիս 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 տեսակի: *void*-ի օգտագործումը

Ինչպես հիշում եք՝ ֆունկցիայի գրելաձևը հետևյալն է՝

**տիպ անուն ( արգումենտ1, արգումենտ 2, ... ) մարմին**

Ինչպես տեսնում եք այն սկսվում է **տիպով**. Բայց ինչպե՞ս վարվել, եթե չենք ուզում, որ ֆունկցիան որևէ արժեք վերադարձնի:

Ենթադրենք՝ ուզում ենք գրել ֆունկցիա, որը էկրանի վրա որևէ տեքստ է տպում: Այդ դեպքում մենք 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.h>

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:

Հիշիր, 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 (Functions) (II)

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

Դեռևս 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: Օրինակ երբ մենք կանչենք նախորդ գլխում քննարկված գումարման ֆունկցիան հետևյալ ձևով՝

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

Այս դեպքում ֆունկցիան կանչելիս մենք 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 );
```


Այս ձևով ֆունկցիան կանչելիս **a** և **b** փոփոխականներում կգրվեն 5 և 3 թվերը, բայց 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:

Բայց որոշ դեպքերում մեզ կարող է հարկավոր լինի կատարել փոփոխություններ արտաքին փոփոխականների հետ, որոնք փոխանցվում են ֆունկցիային՝ որպես արգումենտ: Այս դեպքում պետք է փոփոխականները փոխանցել հասցեով՝ ինչպես արված է հաջորդ օրինակում:

```
// argumentneri poxancum` hasceov  
#include <iostream.h>  
  
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
```

Առաջին նորությունը որ պետք է ուշադրություն գրավի **krknapatik** ֆունկցիայի հայտարարության մեջ տարօրինակ (&) սիմվոլների առկայությունն է. դրանք օգտագործվում են որպեսզի նշել, 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)
    
    krknapatik ( x , y , z );
```

Մեկ այլ ձևով ասած՝ որպես **a**, **b** և **c** արգումենտներ մենք 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 կբերի **y**-ի փոփոխության. նույնը **c**-ի և **z**-ի մասին:

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

Եթե

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

ֆունկցիան հայտարարելիս մենք գրեինք՝

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

(առանց **&** նշանների) ապա արգումենտները փոխանցված կլինեին արժեքով, այլ ոչ թե հասցեով և **a**, **b** կամ **c** փոփոխականները փոփոխելիս **x**, **y** և **z**-ը կմնաին 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.h>

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 մենք կարող ենք ամեն արգումենտի համար նշել լռության արժեքներ: Այդ արժեքները կօգտագործվեն, եթե ֆունկցիան կանչելիս այդ արգումենտները դատարկ թողնվեն: Դա կատարելու համար պարզապես ֆունկցիան հայտարարելիս արգումենտները հավասարացնում ենք անհրաժեշտ արժեքներին: Այժմ եթե ֆունկցիան կանչելիս այդ արգումենտները դատարկ թողնենք, դրանք կփոխարինվեն լռության արժեքներով, իսկ եթե դատարկ չթողնենք, ապա լռության արժեքները կանտեսվեն և կգործի ֆունկցիային արգումենտներ փոխանցելու սովորական օրենքները: Օրինակ՝

```
// lruty an arjeqner
#include <iostream.h>

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)**

մենք նշեցինք միայն մի արգումենտ և մյուսը փոխարինվեց իր լռության արժեքով՝ **2** -ով: Դա կատարվում է քանզի 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)**

Այս դեպքում բոլոր արգումենտները «տեղում են» և լռության արժեքը պարզապես անտեսվում է և գործում են արգումենտի փոխանցման ստանդարտ կանոնները: Արդյունքում ստացվում է **5 (20/4)**.

## Ֆունկցիաների ծանրաբեռնում (Overloading)

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

```
// gerbernva& funkcia
#include <iostream.h>

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 մեկը ընդունում է **int** տիպի երկու փոփոխական, իսկ մյուսը՝ **float** տեսակի երկու փոփոխական: Թարգմանիչը գիտի, թե 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** տեսակի արգումենտ ունեցող ֆունկցիան: Իսկ **float**-ի դեպքում կանչվում է **float** տեսակի 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 անել 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 ամեն 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.h>
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 դարձրու, թե ինչպես է **factorial** ֆունկցիան կանչում ինքն իրեն, բայց միայն այն դեպքում երբ արգումենտը 1-ից մեծ է. այլապես ծրագիրը կընկներ անվերջ բեկուրսիվ 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 մենք հայտարարում էինք վերջում: Եթե ֆունկցիաների նախորդ օրինակներից մեկում **main**-ը բերենք մյուս ֆունկցիաներից առաջ (որոնք կանչվում են **main**-ից) 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 մենք 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: Մրանից 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 պարունկաում ֆունկցիայի մարմինը. այսինքն, բացակայում են { } նշանները և նրանց մեջ գրվող հրամանները:
- Այն վերջանում է կետ-ստորակետով (;):
- Բավարար է նշել միայն արգումենտների տիպերը: Սակայն խորհուրդ է տրվում գրել նաև նրանց անունները՝ ինչպես սովորական ֆունկցիայի հայտարարման մեջ:

## Օրինակ՝

```
// naxatip
#include <iostream.h>
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 տողը **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 մեջ:

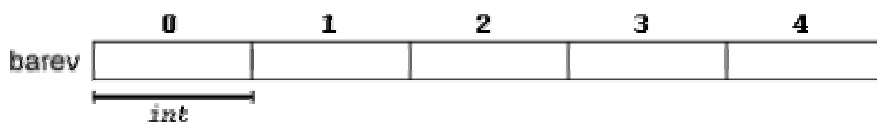
Խորհուրդ է տրվում նախատիպեր սահմանել բոլոր ֆունկցիաների համար, քանզի երբ բոլոր ֆունկցիաների նախատիպերը գրված են միևնույն տեղում, ավելի հեշտ է գտնել մեզ անհրաժեշտ ֆունկցիան: Ֆունկցիաների նախատիպերը սովորաբար գրում են առանձին **վերնագիր** (header) ֆայլերում, այնուհետև դրանք կցում ծրագրին:

## Բաժին 3.1

### Զանգվածներ (Arrays)

Զանգվածը իրենից ներկայացնում է նույն տիպի էլեմենտների (փոփոխականների) խումբ, որոնք հաջորդաբար դասավորված են հիշողության մեջ և որոնք կարող են դիմվել զանգվածի անվանը կցելով համապատասխան էլեմենտի ինդեքսը:

Սա նշանակում է, որ մենք կարող ենք պահել 5 հատ **int** տիպի փոփոխականներ, առանց հայտարարելու 5 հատ՝ իրարից տարբեր անուններով փոփոխականներ: Օրինակ՝ 5 հատ **int** տիպի փոփոխականներ պարունակող *barev* անունով զանգվածը, կարող է ներկայացվել հետևյալ կերպ.



որտեղ յուրաքանչյուր դատարկ ուղղանկյունը իրենից ներկայացնում է զանգվածի էլեմենտ, որը, մեր դեպքում, **int** տիպի է: Այս էլեմենտները ինդեքսավորված (համարակալված) են 0-4, քանի որ զանգվածներում առաջին էլեմենտի ինդեքսը միշտ 0 է՝ անկախ զանգվածի երկարությունից:

Ինչպես և բոլոր մյուս փոփոխականները, զանգվածը պետք է հայտարարված լինի մինչև նրա օգտագործումը: C++-ում զանգվածի տիպիկ հայտարարությունն է.

*տիպ անուն [էլեմենտների քանակ];*

որտեղ *տիպը*՝ զանգվածի էլեմենտների տիպն է, *անունը*՝ զանգվածի անունն է, իսկ *էլեմենտների քանակը*՝ զանգվածի էլեմենտների քանակը: Այսպիսով, որպեսզի հայտարարել 5 էլեմենտանոց **int** տիպի *barev* անունով զանգվածը, պետք է գրել.

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

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

### Զանգվածների սկզբնաթեքավորումը (Initialization of arrays)

Լոկալ տեսանելիության տիրույթում (ֆունկցիայում) զանգված հայտարարելիս, նա ավտոմատ չի սկզբնաթեքավորվելու և նրա պարունակությունը կլինի ոչ բացահայտված, քանի դեռ նրան չվերագրվեն ինչ-որ արժեքներ:

Իսկ գլոբալ տեսանելիության տիրույթում (ցանկացած ֆունկցիայից դուրս) զանգված հայտարարելիս, նրա պարունակությունը ավտոմատ սկզբնաթեքավորվելու է զրոներով: Այսպիսով, եթե մենք գլոբալ տեսանելիության տիրույթում հայտարարենք.

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

ապա *barev* զանգվածի յուրաքանչյուր էլեմենտում գրված կլինի 0.

	0	1	2	3	4
barev	0	0	0	0	0

Զանգված հայտարարելիս, մենք նաև ունենք հնարավորություն նրա յուրաքանչյուր էլեմենտին 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 էլեմենտների քանակը, պետք է համընկնի զանգվածի *էլեմենտների քանակին*: Մեր դեպքում, զանգվածի *էլեմենտների քանակը* 5 է, այդ է պատճառը, որ ձևավոր փակագծերի մեջ գրված էլեմենտների քանակը ևս 5 է:

Դուք կարող եք նաև բաց թողնել զանգվածի *էլեմենտների քանակը*, թողնելով այդ գործը թարգմանիչի վրա: Այդ դեպքում, որպես զանգվածի *էլեմենտների քանակ*, կընդունվի ձևավոր փակագծերի մեջ գրված էլեմենտների քանակը.

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

## Զանգվածների էլեմենտներին դիմումը (Access to the values of an array)

Զանգվածի տեսանելիության ցանկացած կետում մենք կարող ենք կարդալ կամ փոփոխել տրված զանգվածի ցանկացած էլեմենտ, ինչպես և սովորական փոփոխականը: Տեսքը հետևյալն է.

*անուն* [*ինդեքս*]

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

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

Օրինակ, որպեսզի *barev* զանգվածի 3-րդ էլեմենտում գրենք 75, կարող ենք գրել.

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

և, օրինակ, որպեսզի *a* փոփոխականին վերագրենք *barev* զանգվածի 3-րդ էլեմենտի արժեքը, կարող ենք գրել.

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

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

Ուշադրություն դարձրեք նաև այն բանին, որ *barev* զանգվածի 3-րդ էլեմենտը՝ *barev[2]*-ն է, առաջին էլեմենտը՝ *barev[0]*-ն, իսկ վերջին էլեմենտը՝ *barev[4]*-ը:

C++-ում բավականին տարածված սխալ է համարվում զանգվածի գոյություն չունեցող էլեմենտին դիմելը, և քանի որ թարգմանիչը դա, որպես սխալ, չի համարում, ապա ծրագրի աշխատանքի

Ժամանակ դա կարող է հանգեցնել ծրագրի սխալ արդյունքի: Դրա բացատրությունը տրվելու է հետագա դասերում, երբ անցնեք ցուցիչ տիպեր:

Տվյալ պահին, կարևոր է տարբերել իրարից, քառակուսի փակագծերի ([ ])-ի երկու, իրարից տարբեր իմաստները: Առաջինը՝ նրանք օգտագործվում են զանգվածների հայտարարության ժամանակ՝ զանգվածների չափերը որոշելու համար, և երկրորդը՝ օգտագործվում են տրված զանգվածի որևէ էլեմենտին դիմելու համար.

```
int barev[5];           // nor zangvaci haitararutiun (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.h>

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 երկչափանի զանգված.

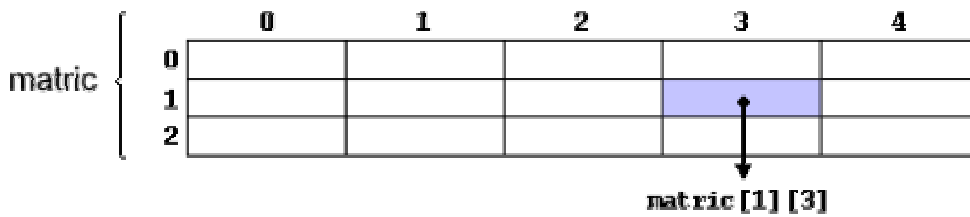
	0	1	2	3	4
0					
1					
2					

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

```
int matric [3][5];
```

և, օրինակ, որպեսզի դիմել 2-րդ տողի 4-րդ էլեմենտին, պետք է օգտագործել հետևյալ արտահայտությունը.

```
matric [1][3]
```



(միշտ հիշեք, որ զանգվածի ինդեքսները սկսվում են 0-ից):

*Բազմաչափ զանգվածները* կարող են լինել ինչպես երկչափ, այնպես էլ եռաչափ, քառաչափ և այլն: Չնայած, սովորաբար, 3 չափանի զանգվածներից մեծ զանգվածներ գրեթե չեն օգտագործվում: Պատկերացրեք միայն, թե ինչքան հիշողություն պետք կգա հետևյալ զանգվածը պահելու համար.

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

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

Բազմաչափ զանգվածները, ուղղակի ստեղծված են ծրագրավորողի հարմարավետության նպատակով, քանի որ, իրականում, մենք 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՝ բազմաչափ.

<pre>// miachap zangvac #include &lt;iostream.h&gt;  #define WIDTH 5 #define HEIGHT 3  int matrix [HEIGHT * WIDTH]; int n,m;  int main () {     for (n=0;n&lt;HEIGHT;n++)     {         for (m=0;m&lt;WIDTH;m++)         {             matrix[n * WIDTH + m]=(n+1) * (m+1);         }     }     return 0; }</pre>	<pre>// bazmachap zangvac #include &lt;iostream.h&gt;  #define WIDTH 5 #define HEIGHT 3  int matrix [HEIGHT][WIDTH]; int n,m;  int main () {     for (n=0;n&lt;HEIGHT;n++)     {         for (m=0;m&lt;WIDTH;m++)         {             matrix[n][m]=(n+1) * (m+1);         }     }     return 0; }</pre>
---	---

Այս ծրագրերից ոչ մեկը չի տպում եկրանին ինչ-որ բան, սակայն երկուսն էլ վերագրում են արժեքներ **matrix** անունով հիշողության բլոկին.

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>0</b>	1	2	3	4	5
<b>1</b>	2	4	6	8	10
<b>2</b>	3	6	9	12	15

Մենք օգտագործեցինք **#define** նախապրոցեսային հրամանը՝ ծրագրի հետագա փոփոխությունները հեշտացնելու նպատակով: Օրինակ, եթե 3\*5 զանգվածի փոխարեն պահանջվի 4\*5 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 արժեքի տեսքով փոխանցել հիշողության մի ամբողջ բլոկ, որպես ֆունկցիայի պարամետր, նույնիսկ, եթե այն հավաքված է զանգվածի մեջ, սակայն կարելի է փոխանցել զանգվածի հասցեն: Դա մոտավորապես նույն բանն է և ավելի արագ ու էֆֆեկտիվ գործողություն է:

Որպեսզի ֆունկցիան, որպես պարամետր կարողանա ընդունել զանգված, անհրաժեշտ է միայն 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.h>

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 մենք 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 [ ] [ երկարություն 1] [ 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 դարձրեք, որ առաջին փակագծերը դատարկ են, իսկ մնացածը՝ ոչ: Դա պետք է միշտ այդպես լինի, որպեսզի թարգմանիչը, ֆունկցիայի մարմնում, կարողանա որոշել յուրաքանչյուր լրացուցիչ չափողականության չափը:

Քիչ փորձ ունեցող ծրագրավորողների մեծամասնությունը, հաճախ սխալվում են ֆունկցիային՝ միաչափ կամ բազմաչափ զանգվածներ փոխանցելուց: Այդ պատճառով, խորհուրդ է տրվում կարդալ բաժին 3.3:

## 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 տա պահել սիմվոլային տողեր: Այդ պատճառով սիմվոլային տողեր պահելու համար օգտագործում են **char**-երի զանգված (հիշենք, որ այս (**char**) տիպը օգտագործվում է մեկ սիմվոլ պահելու համար):

Օրինակ հետևյալ զանգվածը՝

```
char tox [20];
```

կարող է պահել մինչև 20 սիմվոլ: Պատկերացրեք այն այսպես.

**tox**

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Սակայն պարտադիր չէ, որ բոլոր 20 սիմվոլները միշտ սիմվոլ պարունակեն: Օրինակ, **tox**-ը սկզբից կարող է պահել "Barev" տողը, իսկ հետո՝ "Inchpes es?" տողը: Քանի որ սիմվոլային զանգվածները կարող են պահել իրենք չափից ավելի փոքր երկարությամբ տողեր, այդ է պատճառը, որ ընդունված է ավարտել սիմվոլային տողը *զրոյական սիմվոլով*, որի հաստատունը կարող է գրվել **0** կամ **'\0'**:

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

**tox**

B	a	r	e	v	\0													
---	---	---	---	---	----	--	--	--	--	--	--	--	--	--	--	--	--	--

I	n	c	h	p	e	s		e	s	?	\0							
---	---	---	---	---	---	---	--	---	---	---	----	--	--	--	--	--	--	--

Ուշադրություն դարձրեք *զրոյական սիմվոլին* (**'\0'**): Այն ցույց է տալիս, որ սիմվոլային տողը ավարտվել է: Մուգ ներկված քառակուսիները, իրենցից ներկայացնում են անորոշ արժեքներ:

### Տողերի սկզբնարժեքավորում

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

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

Այս դեպքում մենք հայտարարեցինք 6 երկարությամբ սիմվոլային տող (6 էլեմենտանոց **char** -երի զանգված) և սկզբնարժեքավորեցինք այն **Barev** բառը կազմող սիմվոլներով, գումարած զրոյական սիմվոլը՝ **'\0'**-ն:



Սակայն գոյություն ունի նաև սիմվոլային տողերի սկզբնարժեքավորման մեկ այլ եղանակ՝ օգտագործելով հաստատուն տողեր:

Մեր քննարկված նախորդ օրինակներում, մենք արդեն շատ անգամ հանդիպել ենք հաստատուն տողերի: Դրանք սահմանափակված են չափերսներով ("), օրինակ՝

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

արտահայտությունը՝ տողային հաստատուն է:

Ի տարբերություն միավոր չափերսների ('), որոնք միավոր սիմվոլային հաստատուններ են, կրկնակի չափերսները (")" տողային հաստատուններ են: Այն տողերին, որոնք զրված են կրկնակի չափերսների 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 հիշեք. մենք կարող ենք սիմվոլային զանգվածին վերագրել տողային հաստատուն միմիային զանգվածի սկզբնարժեքավորման ժամանակ:

## Արժեքների վերագրումը տողերին

Քանի որ վերագրման *ձախ արժեքը* (lvalue) կարող է լինել միայն զանգվածի էլեմենտ, այլ ոչ թե զանգվածը 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 պրակտիկ մեթոդ չէ: Զանգվածներին և, մասնավորապես, սիմվոլային տողերին արժեքներ վերագրելու համար, կան մի շարք ֆունկցիաներ, ինչպես, օրինակ, **strcpy**-ն: **strcpy** (**string copy**, թարգմանած՝ տողի պատճենում) ֆունկցիան հայտարարված է **cstring** (string.h) գրադարանում և կարող է կանչվել հետևյալ կերպ.

```
strcpy (տող1, տող2);
```

Այն պատճենում է *տոդ2*-ի պատրունակությունը *տոդ1*-ի մեջ: *տոդ2*-ը կարող է լինել զանգված, ցուցիչ կամ տոդային հաստատուն: Հետևյալ արտահայտությունը կվերագրի "**Barev**" հատատուն տոդը **tox** տոդային հաստատունին.

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

Օրինակ.

```
// arjeci veragrum@ toxin
#include <iostream.h>
#include <string.h>

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

**Inchpes es?**

Չմոռանաք ներառել **<string.h>** ֆայլը, երբ օգտագործեք **strcpy** ֆունկցիան:

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

```
// arjeci veragrum@ toxin
#include <iostream.h>

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?**

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

Երբ **cin** -ը օգտագործվում է սիմվոլային տողերի հետ, այն սովորաբար օգտագործվում է իր **getline** մեթոդով, 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* մաքսիմալ երկարությունն է (զանգվածի չափը), իսկ *տարանջատիչը*՝ սիմվոլ է, որը օգտագործվում է մուտքի ավարտը որոշելու համար. լրությամբ այդ սիմվոլը՝ նոր տողի՝ **'\n'**, սիմվոլն է:

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

```
// cin-@ toxeri het
#include <iostream.h>

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@? Ruben
Barev Ruben.
Vorn e dzer sirac tim@? Manchester United
Es sirum em Manchester United-@ evs.
```

Ուշադրություն դարձրեք այն բանին, որ **cin.getline**-ի երկու կանչերի ժամանակ էլ մենք օգտագործել ենք միևնույն իդենտիֆիկատորը (**buffer**): **cin.getline** ֆունկցիայի երկրորդ կանչի ժամանակ, **buffer**-ի պարունակությունը չի ջնջվում, սակայն նոր սիմվոլային տողը սկսվում է գրվել **buffer**-ի առաջին (զրո ինդեքսով) էլեմենտից սկսած: Բայց քանի որ սիմվոլային տողը ավարտվում է զրոյական սիմվոլով, ապա այդ սիմվոլից սկսած բոլոր սիմվոլները անտեսվում են, նույնիսկ եթե դրանք ներկա են:

Եթե դուք հիշում եք “Հաղորդակցություն օգտագործողի հետ” բաժինը, ապա դուք կհիշեք, որ մենք օգտագործում էինք >> օպերատորը ստանդարդ հոսքից ինֆորմացիա կարդալու համար: Այս մեթոդը ևս կարող է օգտագործվել **cin.getline** -ի փոխարեն սիմվոլային տողեր կարդալու համար: Օրինակ մեր ծրագրի այն մասերը, որտեղ մենք պահանջում էինք օգտագործողի կողմից արժեքի մուտքագրում, մենք կարող ենք փոխարինել հետևյալ արտահայտությամբ.

```
cin >> buffer;
```

Սակայն այդ մեթոդը ունի որոշ սահմանափակումներ, որոնք չունի **cin.getline** -ը.

- Այն կարող է օգտագործվել միայն բառեր (ոչ ամբողջ նախադասություններ) կարդալու համար: Պատճառը այն է, որ նրա համար, որպես *տարանջատիչ* ծառայում են դատարկ սիմվոլները, բացատանիշները (space), տաբուլացիաները և նոր տողերը:
- Չի թույլատրվում սահմանել *բուֆերի* երկարությունը: Սա դարձնում է ձեր ծրագիրը ոչ կայուն այն դեպքում, երբ օգտագործողը մուտքագրում է ընդունող զանգվածի չափերից (մեր դեպքում 100) ավելի երկար արտահայտություն:
- Այդ պատճառով, երբ դուք պահանջելու եք **cin** -ից սիմվոլային տողեր, խորհուրդ է տրվում **cin >>** -ի փոխարեն օգտագործել **cin.getline**-ը:

## Տողերի կոնվերտացիան ուրիշ տիպերի

Քանի որ տողի պարունակությունը կարող է իրենից ներկայացնել այլ տիպեր, օրինակ՝ թվեր, ապա հարմար կլինի կարողանալ վերածել տողի պարունակությունը թվային հաստատունի: Օրինակ, տողը կարող է պահել **"1977"**, որը իրենից ներկայացնում է 5 հատ սիմվոլների հաջորդականություն: Սակայն այն այնքան էլ հեշտ չէ վերածել մեկ թվային տիպի: Այդ պատճառով **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 ֆունկցիաները ընդունում են մեկ պարամետր և վերադարձնում են արժեք, ներկայացված տվյալ տիպով (**int**, **long** կամ **float**): Այս ֆունկցիաները, **cin**-ի **getline** մեթոդի հետ միասին, ավելի արդյունավետ են օգտագործողի կողմից մուտքագրված թիվը կարդալու համար, քան **cin>>** մեթոդը.

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

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 յուրաքանչյուրը 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 պահվում մեր փոփոխականը:

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

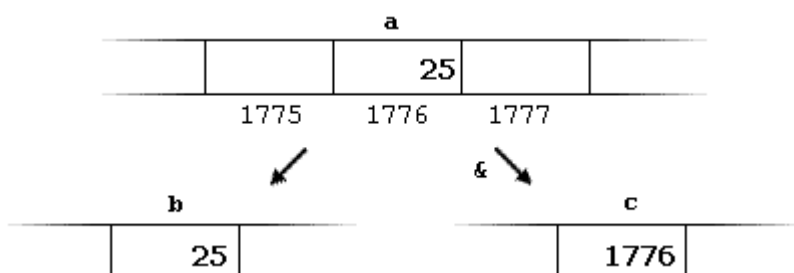
```
a = &b;
```

**a** փոփոխականի մեջ կգրի **b** -ի հասցեն: **b**-ից առաջ դրելով (&) նշան 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՝



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

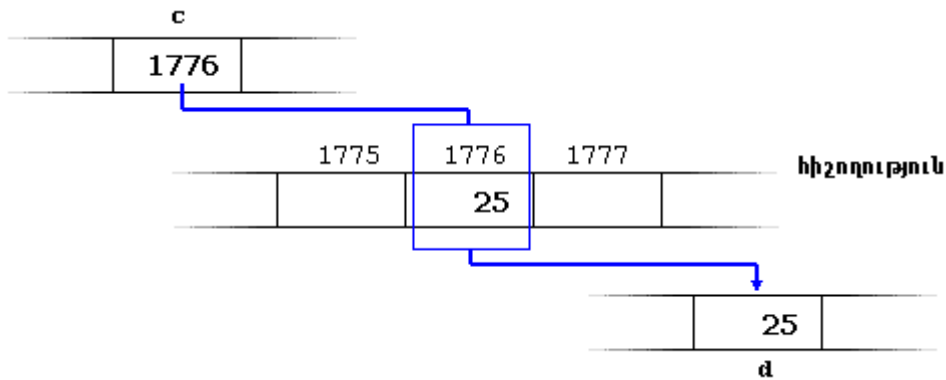
Փոփոխականին, որը պահում է մեկ այլ փոփոխականի հասցե (կոնկրետ օրինակում **c**-ն պահում էր **a**-ի հասցեն), մենք կանվանենք ցուցիչ (**pointer**): Ցուցիչները C++ լեզվի կարևորագույն հասկացություններից են և նրանք ունեն շատ մեծ կիրառություն: Շուտով մենք կտեսնենք, թե ինչպես հայտարարել այդ տիպի փոփոխականներ:

## Ետհասցեավորման (Reference) օպերատոր (\*)

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

```
d = *c;
```

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



Պետք է լավ հասկանալ, որ **c**-ի արժեքը 1776 է, բայց **\*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 քեզ համար պետք է լիովին պարզ լինի, որ եթե նախորդ օրինակում գրված է հետևյալը՝

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

ապա ճիշտ են հետևյալ արտահայտությունները՝

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

## «Ցուցիչ» տիպի փոփոխականների հայտարարում

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

Այսպիսով՝ ցուցիչի հայտարարման գրելաձևը հետևյալն է՝

```
տիպ * ցուցիչ_անուն;
```

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

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

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 ետհասցեավորման օպերատորի հետ, որը նույնպես գրվում է աստղանիշով:

```
// im arachin cucich@
#include <iostream.h>

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

    imcucich = &arjeq1;
    *imcucich = 10;
    imcucich = &arjeq2;
    *imcucich = 20;
    cout << "arjeq1==" << arjeq1 << "/" arjeq2==" << arjeq;
    return 0;
}
```

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

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

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

Ահա մի քիչ ավելի բարդ օրինակ.

```
// eli cucichner
#include <iostream.h>

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**

## Ցուցիչներ և զանգվածներ

Զանգվածների հասկացությունը խիստ կապված է ցուցիչների հասկացության հետ: Զանգվածի նույնարկիչը իրականում համարժեք է զանգվածի առաջին տարրի հասցեին, ինչպես ցուցիչի արժեքը նրա ցույց տվածի հասցեն է: 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 տարբերությունը այն է, որ **p**-ին կարող ենք տալ նոր արժեք, քանի որ այն ցուցիչ է, իսկ **tver**-ը միշտ ցույց կտա 20 թվերից առաջինին: Այսպիսով, ի տարբերություն **p**-ին, 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.h>

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 նշել զանգվածի այն էլեմենտի ինդեքսը, 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 սկզբնաթեքավորում (Initialization)

Ցուցիչ հայտարարելիս կարելի է այն միանգամից սկզբնաթեքավորել՝ նշել այն հասցեն, որին ուզում ենք այն ցույց տալ:

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

սա էկվիվալենտ է՝

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

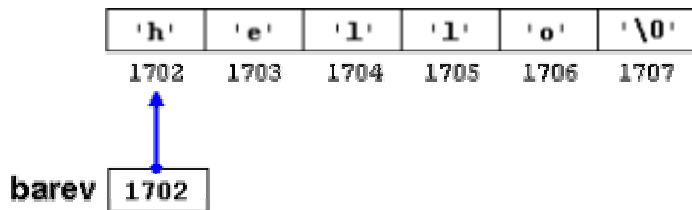
Երբ ցուցիչին վերագրում ենք որևէ արժեք, այդ արժեքը ոչ թե վերագրվում է նրա ցույց տված հասցեով հիշողության տիրույթին, այլ որոշում է թե ինչ հասցեի վրա է ցույց տալիս այդ ցուցիչը: Այս դեպքում (\*) նշանը ոչ մի կապ չունի ետհասցեավորման օպերատորի հետ, այլ պարզապես նշում է, որ հայտարարվում է ցուցիչ: Այսպիսով պետք է զգույշ լինել, որ չխառնել նախորդ գրվածը հետևյալի հետ՝

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

Ինչպես զանգվածների դեպքում, թարգմանիչը հնարավորություն է տալիս ցուցիչները սկզբնաթեքավորել հաստատուններով:

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

Այս դեպքում հիշողության մեջ տեղ է զբաղեցվում "hello" հաստատունը պահելու համար և հիշողության այդ հատվածի առաջին էլեմենտի 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:

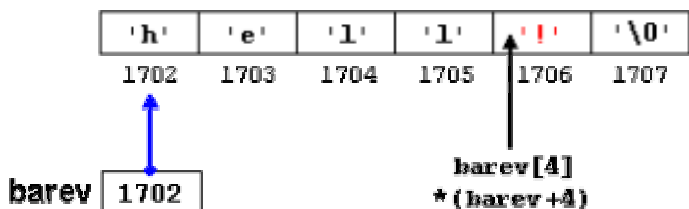
**barev** փոփոխականը ցույց է տալիս սիմվոլների տողի վրա և կարող է օգտագործվել ճիշտ այնպես, ինչպես ցանկացած զանգված: Օրինակ՝ եթե մեր տրամադրությունը փոխվի և մենք որոշենք 'o'-ն փոխարինել '!' -ով ( hell նշանակում է դժողք ), մենք կարող ենք վարվել հետևյալ կերպ.

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

կամ

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

Երկու դեպքում էլ տեղի կունենա սրա պես մի բան՝



## Ցուցիչների թվաբանությունը

Ցուցիչների հետ թվաբանությունը տարբերվում է մյուս ամբողջ տիպերի հետ կատարվող թվաբանությունից: Սկսենք նրանից, որ ցուցիչների համար գոյություն ունեն միայն գումարման և հանման գործողությունները. մյուս գործողությունները պարզապես իմաստ չունեն ցուցիչների աշխարհում: Իսկ գոյություն ունեցող գումարման և հանման գործողությունները տարբեր ցուցիչների համար տարբեր են՝ կախված նրանից, թե ինչ տիպի վրա է ցույց տալիս ցուցիչը:

Ինչպես հիշում եք, ինֆորմացիայի 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՝

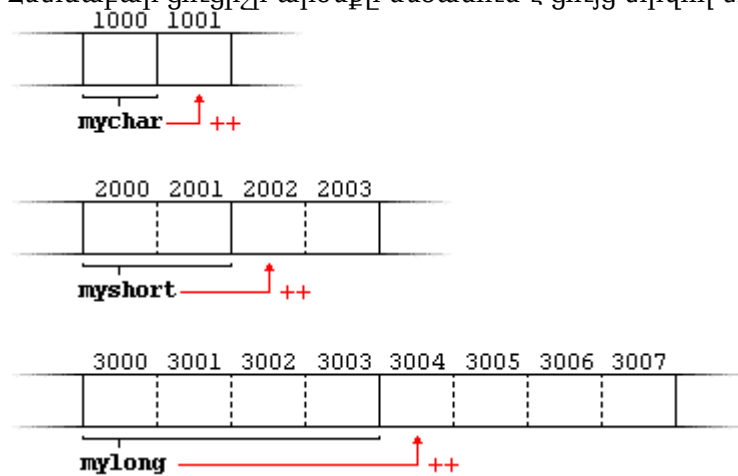
```
char *mychar;
short *myshort;
long *mylong;
```

և մենք գիտենք, որ նրանք համապատասխանաբար ցույց են տալիս հիշողության 1000, 2000 և 3000 հասցեների վրա:

Այսպիսով եթե մենք գրենք.

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

mychar-ը, ինչպես դուք երևի ենթադրում էիք, կպարունակի 1001 արժեքը: Սակայն myshort-ը կպարունակի 2002 արժեքը, իսկ mylong-ը՝ 3004 արժեքը: Պատճառն այն է, որ երբ մենք ցուցիչը մեծացնում ենք 1-ով, մենք նրան ստիպում ենք ցույց տալ նույն տիպի հաջորդ էլեմենտի վրա: Հետևաբար ցուցիչի արժեքը մեծանում է ցույց տրվող տիպի երկարության (բայթերով) չափով:



Սա կիրառելի է և՛ ցուցիչները մեծացնելիս, և՛ փոքրացնելիս: Տեղի կունենար ճիշտ նույնը, եթե գրեինք՝

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

Պետք է հիշել, որ և՛ աճման (++), և՛ նվազման (--) օպերատորները ավելի բարձր նախապատվությամբ գործողություններ են, քան ետհասցեավորման (\*) օպերատորը: Հետևյալ արտահայտությունները կարող են անհասկանալի թվալ

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

Դրանցից առաջինը համարժեք է \* (p++) -ին. սրանում մեծանում է p-ի ցույց տված հասցեն, այլ ոչ թե արժեքը:

Իսկ մյուսը համարժեք է հետևյալին.

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

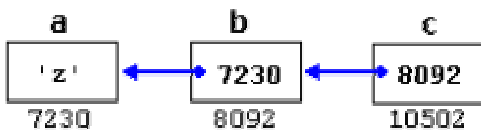
Ինչպես միշտ՝ խորհուրդ ենք տալիս օգտագործել փակագծեր ( ):

## Ցուցիչ ցուցիչի վրա

C++-ը հնարավորություն է տալիս օգտագործել ցուցիչներ, որոնք ցույց են տալիս ցուցիչների վրա, որոնք իրենց հերթին ցույց են տալիս այլ ինֆորմացիայի վրա: Դա անելու համար պարզապես անհրաժեշտ է ավելացնել աստղանիշ (\*)՝ ետհասցեավորման ամեն մակարդակի համար:

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

Եթե համարենք, որ փոփոխականները պահվել են 7230, 8092 և 10502 հասցեներում, ապա դա կարելի է նկարագրել այսպես.



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

## void ցուցիչներ

Ցուցիչի *void* տեսակը յուրահատուկ տեսակ է: *void* տիպի ցուցիչները կարող են ցույց տալ ցանկացած տիպի վրա՝ ամբողջ թվերի, սիմվոլային տողերի և այլն: Միակ սահմանափակումը այն է, որ ցույց տրվող արժեքին չենք կարող անմիջականորեն դիմել (չենք կարող օգտագործել ետհասցեավորման (\*) օպերատորը), քանի որ նրա չափը միշտ անորոշ է: Այս պատճառով մենք միշտ ստիպված կլինենք կատարել տիպի փոփոխում (*type casting*), որպեսզի մեր *void* ցուցիչը դարձնենք ցուցիչ կոնկրետ տիպի վրա, որի հետ կարող ենք հանգիստ աշխատել:

Քննարկենք մի օրինակ՝

```
// amboxch tiv achacnox  
#include <iostream.h>  
  
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:

## Ցուցիչ ֆունկցիայի վրա

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

```

// cucich funkciayi vra
#include <iostream.h>

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 վրա: Այն սկզբնաբժեքավորվել է **hanum** ֆունկցիայով:

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

## Բաժին 3.4

### Դինամիկ հիշողություն (Dynamic memory)

Մինչ այժմ, մեր բոլոր ծրագրերում, մենք վերցնում էինք այնքան հիշողություն, որքան որ անհրաժեշտ էր մեր հայտարարած զանգվածների, փոփոխականների և այլ օբյեկտների համար: Եվ բոլոր դեպքերում հիշողության չափը ֆիքսած էր ծրագրի աշխատանքից առաջ: Այս տիպի հիշողությունը կոչվում է *ստատիկ* հիշողություն: Բայց ինչ, եթե մենք ուզում ենք, օրինակ, որոշել զանգվածի չափը ծրագրի կատարման ընթացքում:

Պատասխանը՝ *դինամիկ հիշողություն*ն է, որի օգտագործման համար ստեղծվել են *new* և *delete* օպերատորները:

#### *new* և *new[]* օպերատորները

Դինամիկ հիշողություն պահանջելու համար, պետք է օգտագործել *new* օպերատորը: *new*-ն հաջորդվում է փոփոխականի *տիպով* և անհրաժեշտության դեպքում, քառակուսի փակագծերի մեջ գրված էլեմենտների քանակով: Նրա տեսքը հետևյալն է.

*ցուցիչ* = **new** *տիպ*

կամ

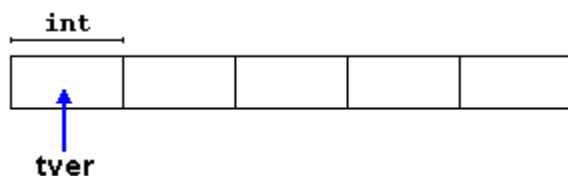
*ցուցիչ* = **new** *տիպ* [*էլեմենտների քանակ*]

Առաջին արտահայտությունը օգտագործվում է այն ժամանակ, երբ հարկավոր է լինում հատկացնել մեկ էլեմենտից կազմված, *տիպ* տիպի դինամիկ հիշողություն, իսկ երկրորդ արտահայտությունը՝ *տիպ* տիպի էլեմենտների բլոկ (զանգված) հատկացնելու համար:

Օրինակ.

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

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



Դուք կարող եք հարցնել, թե ինչու՞մն է տարբերությունը սովորական զանգվածի հայտարարության և ցուցիչին՝ հիշողության վերագրման մեջ: Ամենակարևոր տարբերությունը այն է, որ սովորական զանգվածի չափը պետք է լինի հաստատուն արժեք, որը սահմանափակում է մեր ծրագրի հնարավորությունները, մինչդեռ դինամիկ հիշողության անջատման ժամանակ, մենք կարող ենք պահանջել այնքան հիշողություն, որքան որ պետք է, և անհրաժեշտ չափը արտահայտել փոփոխականներով, հաստատուններով կամ էլ դրանց կոմբինացիայով:

Դինամիկ հիշողությունը կառավարվում է օպերացիոն համակարգի կողմից և կարող է օգտագործվել մի քանի ծրագրերի կողմից: Սակայն կարող է առաջանալ մի դեպք, երբ օպերացիոն համակարգը չունենա ազատ հիշողություն, որպեսզի հատկացնի այն ձեր ծրագրին: Այդ դեպքում **new** օպերատորը կվերադարձնի *զրոյական* (null) ցուցիչ: Այդ պատճառով միշտ խորհուրդ է տրվում ստուգել ցուցիչի արժեքը՝ դինամիկ հիշողություն հատկացնելուց.

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

### ***delete*** օպերատորը

Քանի որ դինամիկ հիշողությունը անվերջ չէ, ապա երբ այն ձեզ արդեն 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 էլեմենտին հատկացված հիշողությունը ջնջելու համար, իսկ երկրորդ արտահայտությունը՝ մի քանի էլեմենտներին (զանգվածին) հատկացված հիշողությունը ջնջելու համար:

```

#include <iostream.h>
#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, այսինքն, կարող է հիշել այնքան թիվ, որքան որ օգտագործողը ցանկանա (իհարկե եթե գոյություն ունենա այդքան ազատ հիշողություն):

**NULL** հաստատուն արժեքը որոշված է C++ի շատ գրադարաններում՝ զրոյական ցուցիչը ներկայացնելու նպատակով: Այն դեպքում, եթե այն որոշված չլինի, դուք ինքներդ կարող եք որոշել այն հետևյալ արտահայտությամբ.

```
#define NULL 0
```

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



## Բաժին 3.5

### Կառուցվածքներ (Structures)

Կառուցվածքային տիպը 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 էլեմենտ1;  
    տիպ2 էլեմենտ2;  
    տիպ3 էլեմենտ3;  
    .  
    .  
}  
օբյեկտի անուն;
```

որտեղ *անունը*՝ կառուցվածքի տիպի անունն է, իսկ *օբյեկտի անունը*՝ տվյալ կառուցվածքային տիպի՝ մեկ կամ մի քանի օբյեկտների (փոփոխականների) անունները:

Կառուցվածքային տիպի անուն գրելը պարտադիր չէ: Բայց այն դեպքում, երբ դուք այն գրեք, այդ կառուցվածքային տիպի մոդելը կդառնա մի նոր տիպ՝ *անուն* անունով: Օրինակ.

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

Մենք սկզբից հայտարարեցինք կառուցվածքային մոդել՝ **apranq**, երկու դաշտերով՝ **anun** և **gin**, որոնց տիպերը տարբեր են: Այնուհետև, մենք 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:

*օբյեկտի անուն* սահմանելը ևս պարտադիր չէ: Այն ուղղակի օգտագործվում է, կառուցվածքային մոդել հայտարարելիս, միանգամից այդ տիպի օբյեկտ հայտարարելու նպատակով: Օրինակ նախորդ օրինակը մենք կարող ենք գրել այսպես.

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

Հիմնականում *անունը* անտեսում են այս օրինակի նման դեպքերում, երբ ներկա է *օբյեկտի անունը*: Սակայն այդ դեպքում, դուք այլևս հնարավորություն չեք ունենա հայտարարել այդ կառուցվածքային տիպի մեկ այլ փոփոխական:

Դուք կարող եք բաց թողնել, նաև, այս երկու պարամետրերը՝ *անունը* և *օբյեկտի անունը*, միաժամանակ, սակայն այդ դեպքում, ձեր գրվածը կլինի անիմաստություն (թե ինչու՝ գուշակեք ինքներդ):

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

Կառուցվածքային տիպի օբյեկտներ հայտարարելուց հետո, մենք 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 են:

Դիտարկենք մեկ այլ օրինակ.

```
// karucvacqneri orinak
#include <iostream.h>
#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 մենք կարող ենք օգտագործել կառուցվածքի էլեմենտները: Ուշադրություն դարձրեք նաև այն բանին, որ **tpel\_usanoxin** ֆունկցիային, որպես պարամետր, փոխանցվում է **usanox** կառուցվածքային տիպի փոփոխական, այլ ոչ կառուցվածքային օբյեկտի դաշտեր:

Ստորև բերված է կառուցվածքների մեկ այլ օրինակ, որը օգտագործում է զանգված.

```
// karucvacqneri orinak
#include <iostream.h>
#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 կառուցվածքների վրա (Pointers to Structures)

Ինչպես և բոլոր այլ տիպերը, կառուցվածքները ևս կարող են ցուցադրված լինել ցուցիչներով: Կանոնները նույնն են, ինչ ցանկացած ֆունկցիոնալ տիպի համար. ցուցիչը պետք է հայտարարված լինի, որպես ցուցիչ կառուցվածքի վրա.

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

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

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

```
pu = &u;
```

Օրինակ.

```
#include <iostream.h>
#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 ծրագիրը օգտագործում է մի նոր գաղափար՝ -> օպերատորը: Սա ետհասցեավորման օպերատոր է, որը օգտագործվում է միայն կառուցվածքների կամ կլասերի վրա ցուցիչների հետ: Այս օպերատորը մեզ թույլ է տալիս չօգտագործել \* սիմվոլը՝ կառուցվածքի դաշտին դիմելիս:

pu->anun

համարժեք է հետևյալին:

(\*pu).anun

Սակայն դուք պետք է տարբերեք (\*pu).anun -ը \*pu.anun արտահայտությունից: \*pu.anun -ը էկվիվալենտ է \*(pu.anun) արտահայտությանը, որը pu կառուցվածքի anun դաշտի ցույց տված արժեքն է (մինչդեռ anun-ը մեր մոտ ցուցիչ չէ):

Հետևյալ ցուցակը ամփոփում է ցուցիչների և կառուցվածքների հնարավոր կոմբինացիաները:

Արտահայտություն	Բացատրություն	Էկվիվալենտ
pu.anun	pu կառուցվածքի anun էլեմենտ	
pu->anun	pu-ի ցույց տված կառուցվածքի anun էլեմենտ	(*pu).anun
*pu.anun	pu կառուցվածքի anun էլեմենտի ցույց տված արժեք	*(pu.anun)

# Կառուցվածքների խտացում

Որպես կառուցվածքի էլեմենտներ կարող են հանդես գալ նաև 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, մենք 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 մի թիվ է:

Այս բաժնում ներկայացված կառուցվածքների հնարավորությունները նույնն են, ինչ C լեզվում: Սակայն C++ լեզվում կառուցվածքների հնարավորությունները ավելի են ընդլայնվել և հասցվել կլասի մակարդակին, միայն այն տարբերությամբ, որ դրանց էլեմենտները համարվում են *public*: Այս ամենին դուք կծանոթանաք բաժնին 4.1-ում:

## Բաժին 3.6

### Օգտագործողի հայտարարած տիպեր

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

### Սեփական տիպերի հայտարարում (typedef).

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

```
C misimvol, miurishsimvol, *ptsimvol1;  
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 որոշ մասի դիմել որպես տարբեր տիպեր, քանի որ նրա բոլոր անդամները պահվում են հիշողության մեջ միևնույն հասցեում: Միավորումները հայտարարվում են ճիշտ այնպես, ինչպես կառուցվածքները, սակայն ունեն բոլորովին այլ հատկություններ:

```
union անուն {
    տիպ1 էլեմենտ1;
    տիպ2 էլեմենտ2;
    տիպ3 էլեմենտ3;
    .
    .
} օբյեկտի_անուն;
```

Միավորման բոլոր էլեմենտները հիշողության մեջ զբաղեցնում են նույն տեղը: Միավորման չափը համընկնում է նրա ամենամեծ էլեմենտի չափի հետ: Օրինակ՝

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

Հայտարարում է 3 էլեմենտ՝

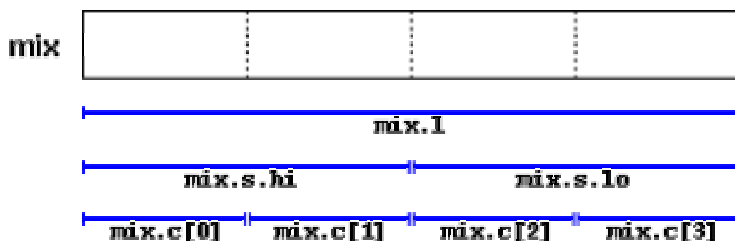
```
imtipper.c
imtipper.i
imtipper.f
```

բոլորը տարբեր տեսակների: Քանի որ նրանք բոլորը պահվում են հիշողության մեջ նույն տեղում, դրանցից մեկի վրա կատարված փոփոխությունը կազդի մյուսների վրա:

Ահա մի 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: Մենք 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 ստրուկտուրայի մեջ մենք տեղադրենք միավորում՝ առանց այդ միավորման համար օբյեկտ նշելու (ձևավոր փակագծերից հետո), ապա այդ միավորումը կլինի անանուն և մենք կկարողանանք անմիջականորեն դիմել նրա էլեմենտներին՝ 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 տարբերությունը այն է, որ առաջինում մենք նշեցինք զինը 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 հիշեցնենք, 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 զինը պետք է լինի կամ դռլարով, կամ դրամով:

## Թվարկումներ (Enumerations) (enum)

Թվարկումները օգտագործվում են, որպեսզի ստեղծել տիպեր, 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;
```

Օրինակ մենք կարող ենք ստեղծել **guyn** անունով մի նոր տիպ, որում կարող ենք պահել տարբեր գույներ.

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

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

```
guyn imguyn;
```

```
imguyn = kapuyt;  
if (imguyn == kanach) imguyn = karmir;
```

Իրականում մեր թվարկած ինֆորմացիան հասկացվում է որպես ամբողջ թվեր: Եթե մենք ոչ մի թիվ չենք նշում, ապա թարգմանիչը 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 արժեքներին, այլ ոչ 0-ից 11, քանի որ մենք **hunvar**-ին տվել էինք 1 արժեքը:

## Բաժին 4.1

### Կլասեր (Ղասային տիպ) (Classes)

Կլասը օգտագործվում է ինֆորմացիան և ֆունկցիաները մեկ միասնական համակարգի մեջ համախմբելու համար: Կլասերը հայտարարվում են **class** բանալի-բառով:

```
class անուն {  
    տեսանելիության նշիչ 1:  
        անդամ1;  
    տեսանելիության նշիչ 2:  
        անդամ2;  
    ...  
} օբյեկտի անուն;
```

որտեղ *անունը*՝ կլասի անունն է (պարտադիր չէ), իսկ *օբյեկտի անունը*՝ տվյալ ղասային տիպի՝ մեկ կամ մի քանի օբյեկտների (փոփոխականների) անունները (պարտադիր չէ): Կլասի մարմինը կարող է պարունակել *անդամներ*, որոնք կարող են լինել կամ փոփոխականներ, կամ ֆունկցիաներ: *Տեսանելիության նշիչները* կարող են լինել՝ **private**, **public** կամ **protected** (պարտադիր չէ): Դրանք ազդում են անդամների վրա հետևյալ կերպ.

- Կլասի **private** անդամներին կարող են դիմել միայն այդ կլասի ուրիշ անդամները կամ *բարեկամ* կլասերը:
- **protected** անդամներին կարող են դիմել միայն այդ կլասի ուրիշ անդամները, *բարեկամ* կլասերը կամ էլ ժառանգող կլասի անդամները:
- **public** անդամներին կարող էք դիմել այն բոլոր տեղերից, որտեղ որ այդ կլասը տեսանելի է:

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

Օրինակ.

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

Հետևյալ կողը հայտարարում է **CRectangle** կլաս և այդ տիպի **rect** անունով օբյեկտ: Այս կլասը 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 դարձրեք նաև այն բանին, որ այստեղ, որպես տիպ է հանդիսանում **CRectangle**-ը, իսկ որպես այդ տիպի օբյեկտ՝ **rect**-ը:

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

*օբյեկտի անուն. անդամ*

ինչպես և կառուցվածքների հետ: Օրինակ.

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

Սակայն մենք չենք կարող օգտագործել **CRectangle** կլասի **x** և **y** փոփոխականները, քանի որ դրանք գտնվում են **private** տեսանելիության տիրույթում և տեսանելի են միայն կլասի սահմաններում, այսինքն՝ կլասի ուրիշ անդամների համար: Օրինակ.

```
// klasneri orinak  
#include <iostream.h>  
  
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 կլասից դուրս կլասի անդամ հայտարարելու համար: Ուշադրություն դարձրեք, որ մենք հայտարարեցինք և որոշեցինք **area()** անդամ ֆունկցիան կլասի մեջ, իսկ **set\_values()** անդամ ֆունկցիան՝ միայն նկարագրեցինք կլասի մեջ և որոշեցինք կլասից դուրս:

Տեսանելիության տիրույթի :: օպերատորը ցույց է տալիս, թե տվյալ անդամ ֆունկցիան, որ կլասին է պատկանում և ապահովում է բոլոր այն հատկությունները, որոնք ներկա կլինեին, եթե այդ անդամ ֆունկցիան որոշված լիներ կլասի մեջ: Օրինակ **set\_values()** ֆունկցիայում մենք օգտագործեցինք **x** և **y** փոփոխականները, որոնք գտնվում են **CRectangle** կլասի **private** տեսանելիության տիրույթում:

Անդամ ֆունկցիան կլասի մեջ կամ կլասից դուրս որոշելու մեջ, միակ տարբերությունը այն է, որ երբ անդամ ֆունկցիան որոշվում է կլասի մեջ, այն թարգմանիչի կողմից համարվում է *inline* (տողաձև) ֆունկցիա, իսկ երբ որոշվում է կլասից դուրս՝ համարվում է սովորական կլասի անդամ ֆունկցիա:

Քանի որ **x** և **y** անդամները մենք հայտարարել ենք **private**, անհրաժեշտ է որոշել **set\_values()** ֆունկցիան, որը կվերագրի արժեքներ այդ անդամներին, քանզի այդ անդամների արժեքները չեն կարող ձևափոխվել ծրագրի մեջ, առանց միջնորդ՝ **set\_values()** ֆունկցիայի: Միգուցե, այս փոքր ծրագրում, այս ձևի էֆեկտիվությունը չի նկատվում, սակայն շատ դեպքերում, մեծ պրոյեկտներ գրելիս, անհրաժեշտ է լինելու չթողնել արժեքների ձևափոխումը առանց միջնորդ ֆունկցիաների:

Կլասի առավելություններից մեկն նաև այն է, որ մենք դրանից կարող ենք հայտարարել մի քանի տարբեր օբյեկտներ: Օրինակ, նախորդ օրինակում մենք կարող ենք հայտարարել **CRectangle** տիպի մեկ այլ՝ **rectb** օբյեկտ.

```
// klasi orinak
#include <iostream.h>

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 դարձրեք այն բանին, որ **rect.area()** ֆունկցիային կանչը, նույն արդյունքը չի թողնում, ինչ **rectb.area()** ֆունկցիային կանչը: Դրա պատճառը այն է, որ այս երկու՝ **rect** և **rectb**, օբյեկտներից յուրաքանչյուրը 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 ֆունկցիաները օբյեկտի հատկություններն են, ի տարբերություն կառուցվածքային ծրագրավորման, որտեղ օբյեկտները ներկայանում են, որպես ֆունկցիայի պարամետրեր: Այս և հաջորդ բաժիններում մենք կքննարկեն այս գաղափարի առավելությունները:

## Կառուցիչներ և փլուզիչներ (Constructors and destructors)

Սովորաբար, օբյեկտի ստեղծման ժամանակ, անհրաժեշտ է լինում սկզբնառժեքավորել փոփոխականներ կամ էլ հատկացնել դիմանիկ հիշողություն, որպեսզի կանխել անորոշ արժեքների վերադարձի դեպքը: Օրինակ, ինչ կկատարվի նախորդ ծրագրում, եթե մենք կանչենք **area()** ֆունկցիան, առանց նախորոք կանչելու **set\_values()** ֆունկցիան: Հավանական է, որ կվերադարձվի անորոշ արժեք, քանի որ **x** և **y** անդամներին երբեք չեն վերագրվել որևէ արժեքներ:

Սա կանխելու նպատակով, կլասը կարող է պարունակել հատուկ ֆունկցիա՝ *կառուցիչ*: *Կառուցիչ* ֆունկցիա հայտարարելու համար, ուղղակի պետք է անվանել անդամ ֆունկցիան կլասի անունով: Այս ֆունկցիան ավտոմատ կանչվելու է տվյալ կլասի նոր օբյեկտներ ստեղծելուց: Օրինակ.

```
// klaseri orinak
#include <iostream.h>

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
```

Իչպես տեսնում եք, այս ծրագրի արդյունքը նույնն է, ինչ նախորդինը: Այս օրինակում, մենք միայն փոխարինեցինք **set\_values** ֆունկցիան կլասի *կառուցիչով*: Ուշադրություն դարձրեք, թե ինչպես են փոխանցվում պարամետրները կառուցիչին՝ նոր օբյեկտների ստեղծման ժամանակ.

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

Դուք նաև կարող եք տեսնել, որ ոչ կլասում՝ կառուցիչի նկարագրման մեջ, և ոչ էլ կառուցիչի որոշման մեջ, գրված չէ վերադարձվող արժեքի տիպը (նույնիս **void** չի գրված): Դա պետք է միշտ այդպես լինի: Կառուցիչը երբեք արժեք չի վերադարձնում, նույնիսկ **void** տիպի, ինչպես ցույց է տրված նախորդ օրինակում:

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

*Փլուզիչը* պետք է ունենա նույն անունը, ինչ կլասը, բայց որը սկսվում է ~ սիմվոլով, և 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.h>

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 ծանրաբեռնում (Overloading constructors)

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

Այն դեպքում, երբ մենք հայտարարենք մի կլաս, բայց չորոշենք որևէ կառուցիչ, թարգմանիչը ավտոմատ կընդունի երկու ծանրաբեռնված կառուցիչները՝ *լռությամբ կառուցիչը* և *պատճենման կառուցիչը*: Օրինակ հետևյալ կլասի 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, թարգմանիչը ավտոմատ կընդունի, որ այն ունի հետևյալ կառուցիչի անդամ ֆունկցիաները.

- **Դատարկ կառուցիչ**

Սա, առանց որևէ պարամետրերի կառուցիչ է, որի մարմինը դատարկ է:

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

- **Պատճենման կառուցիչ**

Սա կլասի 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 հիշեք, որ *դատարկ կառուցիչը* և *պատճենման կառուցիչը* օգտագործվում են միայն այն ժամանակ, երբ տվյալ կլասի համար ոչ մի կառուցիչ չի հայտարարված: Այն դեպքում, երբ հայտարարված է գոնե մեկ կառուցիչ, այս երկու կառուցիչներից ոչ մեկը չի օգտագործվելու:

Օրինակ.

```
// klasi karucichneri canrabernum
#include <iostream.h>

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

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

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



## Ցուցիչներ կլասերի վրա (Pointers to classes)

Ինչպես և բոլոր այլ տիպերը, կլասերը ևս կարող են ցուցադրված լինել ցուցիչներով: Կլասերի վրա ցուցիչներ հայտարարելու տեսքը հետևյալն է.

*Կլասի անուն \* ցուցիչի անուն;*

Օրինակ.

```
CRectangle * prect;
```

ցուցիչ է **CRectangle** կլասի օբյեկտի վրա:

Ինչպես մենք անում էինք կառուցվածքների հետ, այնպես էլ այստեղ, կլասերի դեպքում, որպեսզի դիմենք ցուցիչի ցույց տված կլասի օբյեկտի անդամին, պետք է օգտագործենք -> օպերատորը: Օրինակ.

```
// klasneri vra cucichner
#include <iostream.h>

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 դուք կարող եք կարդալ ցուցիչների և կլասերի օպերատորները (\*, &, ., ->, [ ]), որոնք հանդիպում են օրինակում.

<b>*x</b>	կարդացվում է:	<b>x</b> -ի ցույց տված
<b>&amp;x</b>	կարդացվում է:	<b>x</b> -ի հասցե
<b>x.y</b>	կարդացվում է:	<b>x</b> օբյեկտի <b>y</b> անդամ
<b>(*x).y</b>	կարդացվում է:	<b>x</b> -ի ցույց տված օբյեկտի <b>y</b> անդամ
<b>x-&gt;y</b>	կարդացվում է:	<b>x</b> -ի ցույց տված օբյեկտի <b>y</b> անդամ
<b>x[0]</b>	կարդացվում է:	<b>x</b> -ի ցույց տված առաջին օբյեկտ
<b>x[1]</b>	կարդացվում է:	<b>x</b> -ի ցույց տված երկրորդ օբյեկտ
<b>x[n]</b>	կարդացվում է:	<b>x</b> -ի ցույց տված $(n+1)$ -րդ օբյեկտ

## **struct քանալի-բառով որոշված կլասեր**

C++ լեզվում կառուցվածքը (**struct**), ունի նույն ֆունկցիոնալությունը, ինչ կլասը (**class**), միայն այն տարբերությամբ, որ կառուցվածքի անդամները լռությամբ **public** են, միջդեռ կլասինը՝ **private**:

Բայց քանի որ **class**-ը և **struct**-ը C++ լեզվում ունեն գրեթե նույն ֆունկցիոնալությունը, **struct**-ը սովորաբար օգտագործվում է միայն ինֆորմացիա (փոփոխականներ) պահելու համար, իսկ **class**-ը՝ կլասերի համար, որոնց անդամները և՛ ֆունկցիաներ են, և՛ փոփոխականներ:

## Բաժին 4.2

### Օպերատորների ծանրաբեռնում (Overloading operators)

C++-ը հնարավորություն է տալիս ծանրաբեռնել լեզվի ստանդարդ օպերատորները՝ սահմանելով տարբեր գործողություններ կլասերի համար: Օրինակ

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

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

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

առանց սահմանելու գումարման գործողությունը: Միակ գործողությունը, որը սահմանված է՝ վերագրման գործողությունն է, որը թույլ է տալիս մի տիպի կլասի (կառուցվածքի) վերագրումը, մյուս՝ նույն տիպի կլասին (կառուցվածքին) (լրությամբ պատճենման կառուցիչ):

C++-ում մենք կարող ենք սահմանել այն գործողությունները, որոնք դեռևս որոշված չեն տվյալ կլասի համար (օրինակ գումարման գործողությունը) և նույնիսկ կարող ենք ձևափոխել արդեն որոշված օպերատորները: Ստորև բերված է օպերատորների ցուցակ, որոնք կարող են ծանրաբեռնվել.

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	new	delete		

Օպերատոր ծանրաբեռնելու համար անհրաժեշտ է գրել կլասի անդամ ֆունկցիա, որի անունը սկսվում է **operator** բառով և վերջանում օպերատորի նշանով, որը մենք ուզում ենք ծանրաբեռնել: Տեսքը.

*տիպ operator նշան (պարամետրեր);*

Ստորև բերված է մի օրինակ, որում ծանրաբեռնվում է `+` օպերատորը: Այս ծրագիրը գումարելու է իրար երկու՝ `a(3,1)` և `b(1,2)`, երկչափ վեկտորները: Երկու երկչափ վեկտորների գումարումը, նշանակում է նրանց համապատասխան կոորդինատների գումարումը, 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.h>  
  
class CVector {  
public:  
    int x,y;  
    CVector () {};  
    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** կլասի **operator+** ֆունկցիան պատասխանատու է թվաբանական + օպերատորի ծանրաբոլորման համար: Այն կարող է կանչվել հետևյալ երկու ձևերով.

```

c = a + b;
c = a.operator+ (b);

```

Ուշադրություն դարձրեք նաև այն բանին, որ մենք հայտարարել ենք դատարկ կառուցիչ՝ դատարկ մարմնով.

```
CVector () { };
```

Սա պարտադիր է, քանի որ արդեն գոյություն ունի մեկ այլ կառուցիչ՝

```
CVector (int, int);
```

և *լռությամբ կառուցիչներից* ոչ մեկը չի օգտագործվելու: Դրա համար մենք պետք է հայտարարենք այդպիսի կառուցիչը ինքներս, հակառակ դեպքում հետևյալ արտահայտությունը կհամարվի սխալ՝

```
CVector c;
```

Սակայն դատարկ մարմնով կառուցիչներ ստեղծելը խորհուրդ չի տրվում, քանի որ դրանք չեն բավարարում կառուցիչների մինիմալ պահանջներին՝ փոփոխականների սկզբնաբաժանումը: Մեր դեպքում կառուցիչը թողնում է **x** և **y** փոփոխականներին անորոշ: Ճիշտ կլիներ այդ կառուցիչի փոխարեն գրել

```
CVector () { x=0; y=0; };
```

որը ծրագրի պարզության համար անտեսված է:

Լռությամբ որոշված **վերագրման օպերատորը** պատճենում է պարամետր օբյեկտի (աջ մասում գրված օբյեկտի) բոլոր ոչ ստատիկ ինֆորմացիոն անդամների պարունակությունը ձախ մասում գրված, նույն տիպի օբյեկտի մեջ: Բայց, իհարկե, դուք կարող եք սահմանել այն ձեր ուզած ձևով:

Չնայած նրան, որ պարտադիր չէ, որ ծանրաբեռնված գործողությունները համապատասխանեն մաթեմատիկական գործողություններին կամ դրանց իմաստին, սակայն խորհուրդ է տրվում պահպանել այս պայմանը: Օրինակ, ցանկալի չէ, որ + գործողությունով սահմանվի տարբերության գործողությունը:

Չնայած, որ **operator+** ֆունկցիայի նախատիպը կարող է թվալ ակնհայտ, քանի որ այն վերցնում է օպերատորի աջ մասը, որպես **operator+** ֆունկցիայի պարամետր, սակայն ուրիշ օպերատորների դեպքը այնքան էլ ակնհայտ չէ: Ստորև բերված է ցուցակ, թե ինչպես պետք է հայտարարվեն տարբեր **օպերատոր** ֆունկցիաները (փոխարինեք @ սիմվոլը 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** կլասի օբյեկտ է, **b** –ն՝ **B** կլասի օբյեկտ է, **c** –ն՝ **C** կլասի օբյեկտ է:

Այս ցուցակից դուք տեսնում եք, որ կա օպերատորներ ծանրաբեռնելու երկու եղանակ՝ որպես *անդամ ֆունկցիա* կամ որպես *գլոբալ ֆունկցիա*: Երկրորդ՝ *գլոբալ ֆունկցիա* դեպքը մենք կքննարկենք ավելի ուշ:

## this Բանալի-բառ (The keyword this)

Բանալի-բառ **this**-ը կլասի սահմաններում ցույց է տալիս տվյալ օգտագործվող կլասի օբյեկտի՝ հիշողության մեջ զբաղեցրած հասցեն:

Այն կարող է օգտագործվել, օրինակ, ստուգելու, թե արդյոք փոխանցված պարամետրը հենց ինքը՝ օբյեկտն է, թե ոչ: Օրինակ.

```
// this
#include <iostream.h>

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 մենք կարող էինք գրել **operator=** ֆունկցիան հետևյալ կերպ.

```
CVector& CVector::operator= (const CVector& param)
{
    x=param.x;
    y=param.y;
    return *this;
}
```

Փաստարձն սա լռությամբ կողմն է, որը ավելացվում է մեր կլասին, երբ մենք չենք սահմանում **operator=** անդամ ֆունկցիան:

## Ստատիկ անդամներ (Static members)

Կլասը կարող է պարունակել ստատիկ անդամներ, փոփոխականներ կամ ֆունկցիաներ:

Կլասի ստատիկ անդամները նաև հայտնի են «կլասի փոփոխականներ» անունով, քանի որ նրանց պարունակությունը կախված չէ օբյեկտներից, այլ ֆիքսած է (միևնույնն է) այդ կլասի բոլոր օբյեկտների համար:

Օրինակ գրենք մի ծրագիր, որում կլասի ստատիկ անդամը ցույց տա, թե ինչքան այդ կլասի տիպի օբյեկտ է հայտարարված.

```
// klasi statik andamner
#include <iostream.h>

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 անդամները ունեն նույն հատկությունները, ինչ զլոբալ փոփոխականները, բայց գտնվում են կլասի տեսանելիության տիրույթում: Ընդունված է նաև տալ ստատիկ անդամի նախատիպը (նկարագրությունը) կլասի մեջ, իսկ որոշումը (սկզբնարժեքավորումը)՝ կլասից դուրս՝ զլոբալ տեսանելիության տիրույթում, ինչպես և արված է նախորդ ծրագրում:

Քանի որ ստատիկ անդամը միևնույն փոփոխականն է իր բոլոր կլասի օբյեկտների համար, այն կարող է դիմվել, որպես ցանկացած տվյալ կլասի օբյեկտի անդամ կամ նույնիսկ կլասի անունով (սա ճիշտ է միայն ստատիկ անդամների համար):

```
cout << a.n;  
cout << CDummy::n;
```

Այս երկու կանչերն էլ ցույց են տալիս միևնույն փոփոխականին՝ **CDummy** կլասի **n** ստատիկ փոփոխականին:

Ինչպես մենք նոր կլասի մեջ օգտագործեցինք ստատիկ փոփոխական, այնպես էլ մենք կարող ենք օգտագործել ստատիկ ֆունկցիաներ: Սրանց իմաստը նույնն է՝ սրանք գլոբալ ֆունկցիաներ են, որոնք կանչվում են, իբրև դրանք տվյալ կլասի օբյեկտի անդամ ֆունկցիաներ են: Սակայն ստատիկ ֆունկցիաները կարող են օգտագործել միայն ստատիկ անդամներ, այսինքն, նրանք չեն կարող դիմել տվյալ կլասի ոչ ստատիկ անդամներին, քանի որ դրանք արդեն կախված են օբյեկտից, ինչպես նաև չեն կարող օգտագործել **this** բանալի-բառը:

## Բաժին 4.3

### Կլասերի հարաբերությունները

#### Բարեկամ ֆունկցիաներ (Friend functions)

Բաժին 4.1-ում մենք տեսանք, որ գոյություն ունի կլասի անդամների պաշտպանության երեք աստիճան՝ **public**, **protected** և **private**: Այն դեպքում, երբ կլասի անդամը հայտարարված է, որպես **protected** կամ **private**, այն չի կարող կանչվել տվյալ (որում այն հայտարարված է) կլասի սահմաններից դուրս: Սակայն մենք կարող ենք խախտել այս օրենքը օգտագործելով *friend* բանալի-բառը:

Որպեսզի դրսի ֆունկցիան կարողանա օգտագործել կլասի **private** և **protected** անդամները, մենք պետք է գրենք այդ ֆունկցիայի նախատիպը այն կլասի մեջ, որի **private** և **protected** անդամներին մենք ուզում ենք օգտագործել՝ գրելով **friend** բառը նախատիպի սկզբից: Հետևյալ օրինակում մենք հայտարարում ենք **duplicate** բարեկամ ֆունկցիա:

```
// barekam funkcianer
#include <iostream.h>

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** ֆունկցիայում մենք կարող ենք դիմել **CRectangle** կլասի **width** և **height** փոփոխականներին, որոնք գտնվում են **private** տեսանելիության տիրույթում: Ուշադրություն դարձրեք նաև այն բանին, որ **duplicate()** ֆունկցիան մենք կանչում ենք, ինչպես սովորական գլոբալ ֆունկցիա, այլ ոչ, որպես **CRectangle** կլասի անդամ:



Բարեկամ ֆունկցիաները կարող են ծառայել, օրինակ, երկու տարբեր կլասերի միջև գործողություններ իրականացնելուց: Բարեկամ ֆունկցիայի օգտագործումը համարվում է օբյեկտակողմնորոշված ծրագրավորման գաղափարից դուրս, այդ պատճառով, երբ հնարավոր է, խորհուրդ է տրվում խուսափել նրա օգտագործումից: Օրինակ նախորդ օրինակում ավելի ճիշտ կլիներ ինտեգրացնել `duplicate()` ֆունկցիան `CRectangle` կլասի մեջ:

## Բարեկամ կլասեր (Friend classes)

Ինչպես մենք կարող ենք որոշել բարեկամ ֆունկցիա, այնպես էլ մենք նաև կարող ենք որոշել մի կլաս, որպես մեկ այլ կլասի բարեկամ, թույլ տալով առաջին կլասին դիմել երկրորդ կլասի `private` և `protected` անդամներին:

```
// barekam klas
#include <iostream.h>

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 մենք հայտարարեցինք `CRectangle`-ը, որպես `CSquare` կլասի բարեկամ, թույլ տալով `CRectangle`-ին, դիմել `CSquare` կլասի `private` և `protected` անդամներին, իսկ ավելի կոնկրետ՝ `CSquare::side`-ին, որը ցույց է տալիս քառակուսի կողի երկարությունը:

Այս ծրագրում, դուք կարող եք նաև նկատել մի ինչ-որ նոր բան՝ դատարկ `CSquare` կլասի նախատիպ: Այն պարտադիր է այն պատճառով, որովհետև մենք `CRectangle` կլասի մեջ օգտագործեցինք `CSquare`-

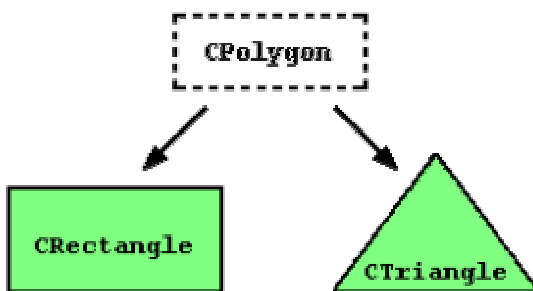
ը, որը **CRectangle**-ի համար դեռևս տեսանելի չէ, քանի որ որոշումը տրվում է **CRectangle** կլասից հետո:

Բարեկամությունը ավտոմատ չի սահմանվում, քանի դեռ մենք ինքներս չսահմանենք այն: Տվյալ օրինակում մենք սահմանեցինք բարեկամություն **CRectangle** և **CSquare** կլասերի միջև, սակայն պետք է զգույշ լինեք այն բանի մեջ, որ մեր դեպքում **CRectangle**-ն է **CSquare**-ի բարեկամ, այլ ոչ **CSquare**-ը՝ **CRectangle**-ի: Այս երկու ասվածների մեջ կա մեծ տարբերություն, քանզի առաջին դեպքում **CSquare**-ի **private** և **protected** անդամներն են հասանելի **CRectangle**-ն, իսկ երկրորդ դեպքում՝ **CRectangle**-ի **private** և **protected** անդամները՝ **CSquare**-ին:

## Կլասերի ժառանգականությունը (Inheritance between classes)

Կլասերի կարևոր հատկություններից է՝ ժառանգականությունը: Այն թույլ է տալիս ստեղծել մի օբյեկտ, որը հիմնված է լինելու մեկ այլ օբյեկտի վրա, և որը պարունակելու է իր սեփական ու ժառանգված օբյեկտի անդամները միասին: Օրինակ, դիցուք մենք ուզում ենք հայտարարել ուղղանկյուն՝ **CRectangle**, և եռանկյուն՝ **CTriangle**, կլասերը: Նրանք երկուսն էլ ունեն ընդհանուր հատկություններ, ինչպես օրինակ նրանք երկուսն էլ կարող են նկարագրվել երկու մեծություններով՝ բարձրությամբ և հիմքով:

Կլասերի աշխարհում **CRectangle** և **CTriangle** կլասերը կարող են նկարագրվել ընդհանուր՝ **CPolygon** կլասով:



**CPolygon** կլասը պարունակելու է այն անդամները, որոնք ընդհանուր են բոլոր բազմանկյունների համար, մեր դեպքում՝ բարձրությունը (**height**) և լայնությունը (**width**):

Այն կլասերը, որոնք ստեղծվում են մեկ այլ կլասերից, ժառանգում են այդ կլասերի տեսանելի անդամները: Դա նշանակում է, որ եթե ժառանգվող կլասը պարունակում է **A** անունով անդամ, և ժառանգող կլասը պարունակում է մի ինչ-որ **B** անդամ, ապա վերջնական կլասը կպարունակի **A** և **B** անդամները:

Որպեսզի մի կլասը ժառանգի մեկ այլ կլասին, պետք է օգտագործենք : օպերատորը ժառանգող կլասի հայտարարության մեջ հետևյալ կերպ.

**class** *կլասի անուն*: **public** *ժառանգվող կլասի անուն*;

Այստեղ **public**-ը կարող է փոխարինվել հետևյալ տեսանելիության նշիչներով՝ **protected** կամ **private**:

## Օրինակ.

```
#include <iostream.h>

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 դուք տեսնում եք այս օրինակից, **CRectangle** և **CTriangle** կլասերի օբյեկտներից յուրաքանչյուրը պարունակում է **CPolygon** կլասի անդամները՝ **width**, **height** և **set\_values()**:

**protected** նշիչը նույնն է, ինչ **private** նշիչը, միայն այն տարբերությամբ, որ երբ մենք ժառանգում ենք մի կլաս, որի անդամները **protected** են, ապա ժառանգված անդամները կարող են օգտագործվել ժառանգող կլասի անդամների կողմից, իսկ **private**-ի դեպքում՝ ոչ: Քանի որ մենք ուզում էինք, որ **CPolygon** կլասի **width** և **height** անդամները կարողանան ձևափոխվել **CRectangle** և **CTriangle** կլասերի անդամների կողմից ևս, այդ պատճառով մենք հայտարարեցինք **CPolygon** կլասի անդամները, որպես **protected**:

Ստորև ամփոփված է, թե ովքեր կարող են դիմել ժառանգվող կլասի անդամներին, դրանց՝ տարբեր տեսանկյունիության տիրույթներում գտնվելու դեպքերում.

Դիմում	public	protected	private
նույն կլասի անդամները	այո	այո	այո
Ժառանգող կլասի անդամները	այո	այո	ոչ
ոչ անդամները	այո	ոչ	ոչ

որտեղ «ոչ անդամները» համարվում են տվյալ կլասերից դուրս գտնվող բոլոր ֆունկցիաները, ինչպիսին է, օրինակ, **main()** -ը:

Մեր օրինակում, անդամները, որոնք ժառանգված են **CRectangle** և **CTriangle** կլասերի կողմից, 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 մենք ժառանգել ենք **CPolygon** կլասը որպես **public**, հետևյալ տողում՝

```
class CRectangle: public CPolygon;
```

Հետևյալ ցուցակը ցույց է տալիս, թե ինչպիսին են լինելու ժառանգվող կլասի անդամների տեսանելիության տիրույթները ժառանգող կլասում՝ անդամների և ժառանգության տեսակի տարբեր դեպքերում.

Ժառանգության տեսակ	Երբ ժառանգվող կլասի անդամը <b>private</b> է	Երբ ժառանգվող կլասի անդամը <b>protected</b> է	Երբ ժառանգվող կլասի անդամը <b>public</b> է
<b>private</b>	անհասանելի է	<b>private</b>	<b>private</b>
<b>protected</b>	անհասանելի է	<b>protected</b>	<b>protected</b>
<b>public</b>	անհասանելի է	<b>protected</b>	<b>public</b>

## Ի՞նչ է ժառանգվում ժառանգվող կլասից

Ժառանգվող կլասի գրեթե յուրաքանչյուր անդամ ժառանգվում է ժառանգող կլասի կողմից, բացի.

- Կառուցիչից և փլուզիչից
- **operator=()** անդամից
- բարեկամներից

Չնայած նրան, որ ժառանգվող կլասի կառուցիչը և փլուզիչը չեն ժառանգվում, սակայն ժառանգվող կլասի լռությամբ կառուցիչը (օրինակ՝ առանց պարամետրերի կառուցիչը) և փլուզիչը միշտ կանչվում են, ժառանգող կլասի նոր օբյեկտներ ստեղծելիս կամ ջնջելիս:

Եթե ժառանգվող կլասը չունի լռությամբ կառուցիչ կամ դուք ուզում եք, որ կանչվի ծանրաբեռնված կառուցիչ, երբ ստեղծվում է ժառանգող կլասի նոր օբյեկտ, դուք կարող եք սահմանել այն ժառանգող կլասի յուրաքանչյուր կառուցիչի հայտարարության ժամանակ հետևյալ կերպ.

*ժառանգող կլասի անուն (պարամետրեր) : ժառանգվող կլասի անուն (պարամետրեր) {}*

Օրինակ.

```
#include <iostream.h>

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** կլասի օբյեկտ հայտարարելուց, կանչվում է **mother** կլասի դատարկ կառուցիչը, իսկ **son** կլասի օբյեկտ հայտարարելուց՝ **mother** կլասի ծանրաբեռնված կառուցիչը:

Տարբերությունը **daughter** և **son** կլասերի կառուցիչների հայտարարության տարբերության պատճառն է.

```
daughter (int a)
son (int a) : mother (a)
```

Նշենք նաև, որ դուք չեք կարող չծանրաբեռնել **son** կլասի կառուցիչը, եթե ուզում եք, որ օգտագործվի **mother** կլասի ծանրաբեռնված կառուցիչը: Այսինքն, չեք կարող գրել.

```
son () : mother (a)
```

քանզի այստեղ *a*-ն չորոշված փոփոխական է: Սակայն դուք կարող եք ծանրաբեռնել **son** կլասի կառուցիչը 2 պարամետրերով, բայց կանչեք **mother** կլասի՝ մեկ պարամետրով ծանրաբեռնված կառուցիչը: Օրինակ.

```
son (int a, int b) : mother (a)
```

## Բազմակի ժառանգականություն (Multiple inheritance)

C++-ում հնարավոր է, որ կլասը ժառանգի մի քանի կլասեր: Դա կատարելու համար, ընդհամենը, պետք է կլասի հայտարարության ժամանակ առանձնացնել ժառանգվող կլասերի անունները ստորակետներոց: Օրինակ, եթե մենք ունենք մի կլաս (**COutput**), որը օգտագործվում է էկրանին ինչ-որ բան տպելու համար, և ուզում ենք, որ մեր **CRectangle** և **CTriangle** կլասերը ժառանգեն այն **CPolygon** կլասի հետ մեկտեղ, պետք է գրենք.

```
class CRectangle: public CPolygon, public COutput { ... };
class CTriangle: public CPolygon, public COutput { ... };
```

Ստորև բերված է լրիվ օրինակը.

```
#include <iostream.h>

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

### Պոլիմորֆիզմ (Polymorphism)

#### Ցուցիչներ ժառանգվող կլասերի վրա

Կլասերի ժառանգականության ամենակարևոր առավելություններից մեկը այն է, որ ժառանգող կլասին ցույց տվող ցուցիչի տիպը համատեղելի է ժառանգվող կլասին ցույց տվող ցուցիչի տիպի հետ: Ստորև բերված է նախորդ բաժիններում քննարկված ուղղանկյան և եռանկյան ծրագիրը, որը օգտագործում է այս հատկությունը:

```
#include <iostream.h>

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** կլասից ծնված օբյեկտներ են, այս վերագրումը ճշմարիտ է:

Միակ սահմանափակումը, որը առկա է **rect**-ի և **trgl**-ի փոխարեն **\*ppoly1**-ի և **\*ppoly2**-ի օգտագործման մեջ, այն է, որ և **\*ppoly1**-ն, և **\*ppoly2**-ն **CPolygon\*** տիպի են, և հետևում է, որ մենք կարող ենք օգտագործել միայն այն անդամները, որոնք **CRectangle** և **CTriangle** կլասերը ժառանգում են **CPolygon** կլասից: Այդ է պատճառը, որ մենք **area()** ֆունկցիան կանչելուց, չօգտագործեցինք **\*ppoly1** և **\*ppoly2** օբյեկտները: Որպեսզի մենք կարողանանք **CPolygon** կլասին



ցույց տվող ցուցիչներով կանչել `area()` ֆունկցիան, անհրաժեշտ է, որ այդ ֆունկցիան հայտարարված լինի `CPolygon` կլասի մեջ:

## Վիրտուալ անդամներ (Virtual members)

Վիրտուալ անդամները, այն անդամ ֆունկցիաներն են, որոնք կարող են վերահայտարարվել ժառանգող կլասում: Այն դեպքում, երբ մենք, ժառանգվող կլասի վրա ցույց տվող ցուցիչով, դիմենք ժառանգող կլասի օբյեկտի անդամին, որը ժառանգվող կլասում հայտարարված էր որպես `virtual`, ապա ժառանգող կլասում այդ ֆունկցիայի առկայության դեպքում, կկանչվի այդ ֆունկցիան:

`virtual` անդամ հայտարարելու համար, անհրաժեշտ է ժառանգվող կլասում՝ անդամի հայտարարության սկզբում, ավելացնել `virtual` բառը:

Դիտարկենք հետևյալ օրինակը.

```
// virtual members
#include <iostream.h>

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 կլասերը (**CPolygon**, **CRectangle** և **CTriangle**) ունեն միևնույն անդամները՝ **width**, **height**, **set\_values()** և **area()**:

**area()** անդամը հայտարարված է **virtual** այն պատճառով, որովհետև այն վերահայտարարվում է ժառանգող կլասերում: Եթե ուզում եք, դուք կարող եք ստուգել, որ եթե ջնջեք **virtual** բառը և աշխատացնեք ծրագիրը, ապա արդյունքը բոլոր երեք դեպքերում կլինի 0՝ 20,10,0 –ի փոխարեն: Դա այդպես կլինի 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 իմաստը այն է, որ այն թողնում է ժառանգող կլասում հայտարարել մի անդամ, որի անունով անդամ արդեն հայտարարված է ժառանգվող կլասում, և երբ այդ անդամը կանչվում է ժառանգվող կլասին ցույց տվող ցուցիչով, ժառանգվող կլասի անդամի փոխարեն կանչվում է ժառանգող կլասում վերահայտարարված անդամը (Տես վերևի օրինակը):

Չնայած, որ **area()** ֆունկցիան վիրտուալ ֆունկցիա է, մենք կարողացանք հայտարարել **CPolygon** տիպի մի օբյեկտ և կանչել նրա այն **area()** ֆունկցիան, որը միշտ վերադարձնում է 0:

## Աբստրակտ կլասեր (Abstract base classes)

Սովորական աբստրակտ կլասերը, մեր նախորդ օրինակում բերված **CPolygon** կլասի նման կլասեր են, միայն այն տարբերությամբ, որ մենք, նախորդ օրինակում, **CPolygon** կլասի օբյեկտների համար որոշեցինք **area()** ֆունկցիան, մինչդեռ *աբստրակտ կլասում*, մենք միայն պետք է թողնեինք այս ֆունկցիայի նախատիպը՝ նրան կցելով **=0** արտահայտությունը:

Այսպիսով մեր **CPolygon** կլասը կարող էր լինել.

```
class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area (void) =0;
};
```

Ուշադրություն դարձրեք, որ մենք կցեցինք **=0**-ն՝ **virtual int area (void)** արտահայտությանը, նրա որոշման փոխարեն: Այս տիպի ֆունկցիաները կոչվում են *մաքուր վիրտուալ ֆունկցիաներ*, և բոլոր կլասերը, որոնք պարունակում են *մաքուր վիրտուալ ֆունկցիաներ*, համարվում են *աբստրակտ կլասեր*:

Աբստրակտ կլասի կարևորագույն հատկությունը այն է, որ նրա օբյեկտներ ստեղծվել չեն կարող, այլ կարող են ստեղծվել միայն ցուցիչներ այդ կլասի վրա: Այսպիսով ստացվում է, որ հետևյալ հայտարարությունը՝

```
CPolygon poly;
```

սխալ է աբստրակտ կլասերի համար: Բայց ճիշտ են հետևյալ հայտարարությունները.

```
CPolygon * ppoly1;
CPolygon * ppoly2;
```

Դա այդպես է, քանի որ *մաքուր վիրտուալ ֆունկցիան*, ինչպիսին այն պարունակում է, որոշված չէ, իսկ կլասի օբյեկտներ կարելի է հայտարարել միայն այն ժամանակ, երբ նրա բոլոր անդամները որոշված են: Սակայն կարելի է հայտարարել ցուցիչ, որը ցույց է տալիս ժառանգող կլասի օբյեկտին, որում հայտարարված է այդ ֆունկցիան:

Օրինակ.

```
// virtual andamner
#include <iostream.h>

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

Եթե դուք կարդաք ծրագիրը, դուք կնկատեք, որ մենք կարող ենք դիմել տարբեր կլասերի օբյեկտներին՝ օգտագործելով հատուկ տիպի ցուցիչ՝ **CPolygon\***: Սա շատ օգտակար բան է: Պատկերացրեք, այժմ մենք կարող ենք ստեղծել **CPolygon** կլասի անդամ ֆունկցիա, որը կկարողանա էկրանին տպել **area()** ֆունկցիայի արդյունքը՝ անկախ ժառանգող կլասից.

```

// virtual andamner
#include <iostream.h>

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 իրենից ներկայացնում է այն օբյեկտի վրա ցույց տվող ցուցիչ, որի ծրագիրը աշխատացվում է:

## Բաժին 5.1

### Կաղապարներ (templates)

#### Ֆունկցիաների կաղապարներ

Կաղապարները հնարավորություն են տալիս ստեղծել ընդհանուր ֆունկցիաներ, որոնք կարող են աշխատել ցանկացած տիպի ինֆորմացիայի հետ: Առանց կաղապարների մենք նույն առաջադրանքը կատարելու համար ստիպված կլինեինք ստեղծել գերբեռնված ֆունկցիաներ՝ ամեն 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 ձևով և ունեն ճիշտ նույն հատկությունները:

Օրինակ, որպեսզի ստեղծել ֆունկցիայի կաղապար, որը կվերադարձնի իրեն, որպես արգումենտ տրված երկու օբյեկտներից մեծագույնը, մենք կարող ենք գրել.

```
template <class GenericType>  
GenericType GetMax (GenericType a, GenericType b) {  
    return (a>b?a:b);  
}
```

Առաջին տողում մենք ստեղծում ենք ինֆորմացիայի տիպի կաղապար, որը անվանում ենք **GenericType**: Դրան հաջորդող ֆունկցիայում **GenericType**-ը դառնում է լիովին վավեր ինֆորմացիայի տիպ: Այդ տիպը մենք օգտագործում ենք որպես **a** և **b** փոփոխականների տիպ, ինչպես նաև **GetMax** ֆունկցիայի վերադարձվող արժեքի տիպ:

**GenericType**-ը, սակայն, ոչ մի կոնկրետ տիպ չի ներկայացնում: **GetMax** ֆունկցիան կարող ենք կանչել ցանկացած վավեր ինֆորմացիայի տիպով: Այդ ժամանակ **GenericType**-ը կփոխարինվի այդ տիպով: Կաղապարային ֆունկցիաները կանչվում են հետևյալ կերպ:

*ֆունկցիա <տիպեր> (արգումենտներ);*

Օրինակ՝ որպեսզի կանչել **GetMax** ֆունկցիան և համեմատել երկու **int** արժեքներ, կարող ենք գրել.

```
int x,y;  
GetMax <int> (x,y);
```

Այսպես **GetMax** ֆունկցիան կկանչվի՝ իրեն դրսևվորելով ճիշտ այնպես, ինչպես կդրսևվորեր, եթե **GenericType**-ի փոխարեն բոլոր տեղերում գրեինք **int**:

## Ահա մի օրինակ՝

```
// function template
#include <iostream.h>

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 մենք հայտարարված ընդհանուր տիպին տվեցինք **T** անուն **GenericType**-ի փոխարեն, քանզի այն ավելի կարճ է և կադապարներում ամենաշատը օգտագործվող նույնարկիչն է: Դրա փոխարեն կարող էինք օգտագործել ցանկացած վավեր նույնարկիչ):

Վերևում բերված օրինակում մենք օգտագործեցինք միևնույն **GetMax()** ֆունկցիան **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. այն տիպի օբյեկտ, որը կգրենք <> նշանների միջև՝ ֆունկցիան կանչելիս:

Կոնկրետ այս դեպքում (երբ **T** տիպը օգտագործվում է, որպես **GetMax** ֆունկցիայի արգումենտ) թարգմանիչը կարող է ինքնուրույն որոշել, թե ինչ տիպ է փոխանցվել ֆունկցիային՝ առանց նշելու **<int>** կամ **<long>** պարամետրերը: Այսպիսով՝ կարող էինք 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 կտա նույն արդյունքը:

```
// function template II
#include <iostream.h>

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

Ուշադրություն դարձրեք, թե ինչպես այս դեպքում **main()** ֆունկցիայից կանչեցինք **GetMax()** ֆունկցիան՝ առանց <> սիմվոլների միջև նշելու, թե ինչ տիպ ենք պահանջում: Դա ֆունկցիայի ամեն կանչի համար որոշում է թարգմանիչը:

Քանի որ մեր կադապարային ֆունկցիան ունի միայն մի ինֆորմացիայի ընդհանուր տիպ (**class T**), և նրա ընդունած երկու արգումենտներն էլ նույն տիպի են, մենք չենք կարող այդ կադապարային ֆունկցիան կանչել երկու տարբեր տիպի արգումենտներով:

```
int i;
long l;
k = GetMax (i,l);
```

Սա սխալ կլինի, քանի որ մեր ֆունկցիան սպասում է նույն տիպի երկու արգումենտ:

Մենք նաև կարող ենք ստեղծել ֆունկցիաներ, որոնք կունենան մեկից ավել 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 տիպի արգումենտներ և վերադարձնում է առաջին արգումենտի տիպի (**T**) արժեք: Վերևում բերված հայտարարումից հետո մենք կարող ենք ֆունկցիան կանչել հետևյալ ձևով:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

կամ պարզապես

```
i = GetMin (j,l);
```

չնայած՝ **j**-ն և **l**-ն տարբեր տիպերի են:

## Կլասերի կադապարներ

Մենք նաև ունենք հնարավորություն ստեղծել կադապարային կլասեր: Դրանք կարող են ունենալ անդամներ, որոնց տիպը չի նշվում կլասի ստեղծման ժամանակ: Օրինակ՝

```
template <class T>
class pair {
    T values [2];
public:
    pair (T first, T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

Այս կլասը հնարավորություն է տալիս պահել ցանկացած՝ գոյություն ունեցող տիպի երկու օբյեկտ: Օրինակ եթե ուզում ենք հայտարարել այս կլասի օբյեկտ, որը պիտի պահի երկու `int` տիպի էլեմենտ, որոնց արժեքները **115** և **36** են, կարող ենք գրել.

```
pair<int> myobject (115, 36);
```

Նույն կլասի միջոցով կարող ենք ստեղծել նաև օբյեկտներ, որոնք կարող են պահել ցանկացած տիպի երկու օբյեկտ՝

```
pair<float> myfloats (3.0, 2.18);
```

Վերևում բերված օրինակում միակ անդամ ֆունկցիան գրեցինք միանգամից՝ կլասի հայտարարման մեջ: Եթե ֆունկցիան առանձնացնում ենք կլասից, ապա այն ամպայման պիտի սկսի

**template <...>-ով:**

```
// class templates
#include <iostream.h>

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



Ուշադրություն դարձրեք, թե ինչպես է սկսվում **getmax** ֆունկցիայի որոշումը՝

```
template <class T>
T pair<T>::getmax ()
```

## Հատուկ կադապարներ

Հնարավոր է ստեղծել կադապարներ, որոնք կարող են ունենալ տարբեր հատկություններ՝ կախված կադապարի ընդհանուր տիպերից: Օրինակ մենք կարող ենք ուղենալ վերևում բերված **pair** կլասի մեջ սահմանել մի ֆունկցիա, որը կվերադարձնի տրված թվերի բաժանումից ստացված մնացորդը: Այդ ֆունկցիան պետք է վերադարձնի մնացորդը միայն **int** տեսակի դեպքում, իսկ մյուս տիպերի դեպքում վերադարձնի **0** : Դա կարելի է անել հետևյալ կերպ.

```
// Template specialization
#include <iostream.h>

template <class T>
class pair {
    T value1, value2;
public:
    pair (T first, T second)
        {value1=first; value2=second;}
    T module () {return 0;}
};

template <>
class pair <int> {
    int value1, value2;
public:
    pair (int first, int second)
        {value1=first; value2=second;}
    int module ();
};

template <>
int pair<int>::module() {
    return value1%value2;
}

int main () {
    pair <int> myints (100,75);
    pair <float> myfloats (100.0,75.0);
    cout << myints.module() << '\n';
    cout << myfloats.module() << '\n';
    return 0;
}
```

25  
0

Ինչպես տեսնում եք կոդի մեջ հատուկ կադապարը հայտարարվում է հետևյալ կերպ.

```
template <> class կլասի_անուն <տիպ>
```

Հատուկ կադապարը նույնպես կլասի կադապար է. այդ պատճառով հայտարարությունը պիտի սկսի **template <>**-ով: Եվ իսկապես, մենք հատուկ կադապարը հայտարարում ենք ոչ թե ցանկացած, այլ խիստ որոշակի տիպի համար, այդ պատճառով **<>** սիմվոլների միջև ոչ մի ընդհանուր տիպ գրել չի կարելի: Կլասի անունից հետո պիտի նշենք այն տիպը, որի դեպքում պիտի գործի այդ կադապարը:

Պետք է հիշել, որ հատուկ կաղապարը ընդհանուր կաղապարից ոչ մի անդամ չի ժառանգում (նախորդ օրինակում հատուկ կաղապարի համար ստիպված էինք սահմանել կառուցիչ՝ չնայած, որ այն ոչնչով չի տարբերվում ընդհանուր կաղապարի կոնստրուկտորից):

## Կաղապարի արգումենտներ

Կաղապար հայտարարելիս մենք օգտագործում ենք **class** կամ **typename** բանալի-անունները, որպեսզի կաղապարին տիպեր փոխանցենք, սակայն կաղապարին կարելի է փոխանցել ցանկացած հաստատուն, ինչպես արված է օրինակում.

```
// array template
#include <iostream.h>

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 հետ:

Ահա կաղապարների հայտարարման մի քանի օրինակ՝

```
template <class T> // amenahachax patahox@. miak tipayin parametr.

template <class T, class U> // erku tipayin parametrer.

template <class T, int N> // class yev amboxch tiv.

template <class T = char> // lrutyan arjeqov.

template <int Tfunc (int)> // funkcia` vorpes parametr.
```

## Կադապարներ և մի քանի ֆայլերով պրոյեկտներ

Թարգմանիչի տեսակետից նայելիս կադապարները սովորական կլասեր և ֆունկցիաներ չեն. նրանք թարգմանվում են միայն այն դեպքում, երբ կա համապատասխան կանչ: Կանչի պատահելու դեպքում թարգմանիչը ստեղծում է համապատասխան ֆունկցիա/կլաս:

Մեծ պրոյեկտներ ստեղծելիս ծրագրի կողը բաժանում են տարբեր ֆայլերի միջև: Այս դեպքում սովորաբար կլասերի/ֆունկցիաների ինտերֆեյսը և իրագործումը տարբեր ֆայլերի մեջ են գրվում: Բոլոր ֆունկցիաների նախատիպերը գրվում են "header" ֆայլերի մեջ (սովորաբար `.h` վերջավորությամբ), իսկ ֆունկցիաների որոշումները՝ `c++` կոդի առանձին ֆայլում:

Կադապարների մակրոյանման աշխատանքը որոշակի սահմանափակումներ է առաջացնում բազմաֆայլ պրոյեկտների դեպքում: Ֆունկցիայի/կլասի կադապարի հայտարարումը և որոշումը չեն կարող լինել տարբեր ֆայլերի մեջ:

## Բաժին 5.2

### Նախաթարգմանիչի հրամաններ (Preprocessor directives)

Նախաթարգմանիչի հրամանները կատարվում են ոչ թե ծրագրի այլ նախաթարգմանիչի կողմից: Նախաթարգմանիչը աշխատեցվում է այն ժամանակ, երբ մենք թարգմանիչին հրամայում ենք թարգմանել կոդը: Նախաթարգմանիչը ստուգում է կոդը, կատարում իրեն տրված հրամանները և սխալ առաջանալու դեպքում կանգնեցնում է թարգմանման պրոցեսը:

Նախաթարգմանիչի բոլոր հրամանները պետք է գրվեն առանձին տողի վրա, և նրանցից հետո պետք է ղնել (;) նշան:

#### #define

Այս գրքի սկզբում մենք արդեն խոսեցել ենք նախաթարգմանիչի **#define** հրամանի մասին: Դրա միջոցով մենք ստեղծում ենք *նշանակված փոփոխականներ* և *մակրոհրամաններ*: Գրելաձևը հետևյալն է:

```
#define անուն արժեք
```

Նախաթարգմանիչը ամեն անգամ կոդի մեջ հանդիպելով *անուն* բառը՝ 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 մասը անտեսել՝ կախված ինչ որ պայմաններից: Պետք է հիշել, որ սա կատարվու է ծրագիրը թարգմանելիս այլ ոչ թե աշխատացնելիս:

**#ifdef**-ը հնարավորություն է տալիս ծրագրի մի մասը թարգմանել միայն այն դեպքում, երբ այն անունը, որը տրվել է նրան որպես արգումենտ, հանդիսանում է նշանակված հաստատուն՝ անկախ արժեքից:

```
#ifdef անուն
// ծրագրի կոդը գրվում է այստեղ
#endif
```

Օրինակ՝

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

Այս դեպքում **char str[MAX\_WIDTH];** տողին թարգմանիչը «ուշադրություն է դարձնում» միայն այն դեպքում, եթե **MAX\_WIDTH** -ը նշանակված հաստատուն է: Եթե այն նշանակված հաստատուն չէ, այդ տողը անտեսվում է (ծրագրի մեջ չի պարունակվում):

**#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* = *else if*) հրամաններից հետո գրված կոդը թարգմանվում է միայն որոշակի պայմանների դեպքում: Սրանք կարող են աշխատել միայն 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 դարձրեք, թե ինչպես է **#if**, **#elsif** և **#else** շարքը ավարտվում **#endif**-ով:

## #line

Երբ մենք հրամայում ենք թարգմանել ծրագիրը և թարգմանման ընթացքում որևէ սխալ է առաջանում, թարգմանիչը մեզ ցույց է տալիս այն ֆայլի անունը և տողի համարը, որում տեղի է ունեցել սխալը:

**#line** հրամանը մեզ հնարավորություն է տալիս փոփոխել երկուսն էլ՝ տողերի համարակալումը ինչպես նաև ֆայլի անունը, որը մեզ ցույց կտրվի սխալ առաջանալու դեպքում: Գրելաձևը հետևյալն է՝

```
#line տողի_համար "ֆայլի_անուն"
```

Որտեղ *տողի\_համարը* 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 տողի վրա:

## **#error**

Այս հրամանին հանդիպելիս նախաթարգմանիչը կանգնեցնում է թարգմանման պրոցեսը՝ որպես սխալ վերադարձնելով իրեն, որպես արգումենտ 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: Կա կցվելիք ֆայլը նշելու երկու եղանակ.

```
#include "ֆայլ"  
#include <ֆայլ>
```

Այս երկու ձևերի միակ տարբերությունը այն է, որ առաջին դեպքում նախաթարգմանիչը նշված ֆայլը փնտրում է նույն կատալոգում, որում գտնվում է ներկա թարգմանվող ֆայլը: Մյուս դեպքում նախաթարգմանիչը ֆայլը փնտրում է հատուկ կատալոգներում, որտեղ գտնվում են ստանդարտ գրադարանները:

## **#pragma**

Սա օգտագործվում է թարգմանիչին հատուկ հրամաններ տալու համար: Այդ հրամանները տարբեր թարգմանիչների համար տարբեր են: Դրանց ծանոթանալու համար կարդացեք ձեր թարգմանիչի ձեռնարկը:

## Բաժին 6.1

### Մուտք/Ելք ֆայլերի հետ

C++-ը մեզ հնարավորություն է տալիս ֆայլերի հետ մուտք/ելք կատարել՝ օգտագործելով հետևյալ կլասերը.

- **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 կլասերի օբյեկտների հետ սովորաբար առաջինը կատարվող 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:

Որպեսզի որևէ ֆայլ բացել հոսքի օբյեկտից, օգտագործում ենք նրա անդամ **open()** ֆունկցիան.

```
void open (const char * ֆայլի_անուն, openmode բացման_ձև) ;
```

որտեղ *ֆայլի\_անունը* սիմվոլային տող է, որը ներկայացնում է ֆայլի անունը և, անհրաժեշտության դեպքում, հասցեն ("C:\file.txt"): Իսկ *բացման\_ձևը* հետևյալ դրոշակների կոմբինացիա է՝

<b>ios::in</b>	Ֆայլը բացել կարդալու համար
<b>ios::out</b>	Ֆայլը բացել գրելու համար
<b>ios::ate</b>	Սկզբնական դիրքը՝ ֆայլի վերջում
<b>ios::app</b>	Բոլոր փոփոխությունները գրվում են սկսած ֆայլի վերջից
<b>ios::trunc</b>	Եթե ֆայլը գոյություն ունի, այն մաքրվում է
<b>ios::binary</b>	Երկուական ռեժիմ

Այս դրոշակները կարելի է կոմբինացնել (միանգամից ընտրել մի քանի հատը)՝ օգտագործելով բիթային ԿԱՄ օպերատորը (**|**): Օրինակ՝ եթե մենք ուզում ենք բացել "orinak.bin" ֆայլը և նրա մեջ ինֆորմացիա գրել երկուական ռեժիմում, ապա պետք է կանչենք **open** ֆունկցիան՝ հետևյալ ձևով.

```
ofstream file;  
file.open ("orinak.bin", ios::out | ios::app | ios::binary);
```

Բոլոր երեք կլասերի (**ofstream**, **ifstream** and **fstream**) անդամ **open** ֆունկցիաները ունեն լռության բացման ձև: Կլասերից յուրաքանչյուրի համար այն յուրահատուկ է.

կլաս	լռությամբ բացման ձև
<b>ofstream</b>	<code>ios::out   ios::trunc</code>
<b>ifstream</b>	<code>ios::in</code>
<b>fstream</b>	<code>ios::in   ios::out</code>

Լռության արժեքը օգտագործվում է միայն այն դեպքում, երբ **open** ֆունկցիան կանչելիս չի նշվել բացման ձև. հակառակ դեպքում լռության արժեքը անտեսվում է:

Ինչպես ասեցինք, առաջին գործողությունը, որ արվում է **ofstream**, **ifstream** և **fstream** կլասերի օբյեկտների հետ, ֆայլ բացելն է. այդ պատճառով այս կլասերը ունեն կոնստրուկտոր, որը միանգամից կանչում է **open** անդամ-ֆունկցիան: Այս ձևով մենք կարող էինք հայտարարել նախորդ օբյեկտը՝ պարզապես գրելով

```
ofstream file ("orinak.bin", ios::out | ios::app | ios::binary);
```

Երկու եղանակն էլ կունենան նույն ազդեցությունը:

Դուք կարող եք ստուգել՝ արդյոք ֆայլի բացման պրոցեսը նորմալ է անցել՝ օգտագործելով **is\_open()** անդամ-ֆունկցիան.

```
bool is_open();
```

Սա վերադարձնում է բուլեան տիպ՝ **true**, եթե օբյեկտը նորմալ կապվել է ֆայլի հետ, 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()** անդամ-ֆունկցիան, որը ավարտում է ֆայլի հետ աշխատանքը և փակում ֆայլը: Դրա կանչը շատ հեշտ 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, դեստրուկտորը ավտոմատաբար կանչում է **close** անդամ-ֆունկցիան:

## Ֆայլերի հետ աշխատանք տեքստային ռեժիմում

**ofstream**, **ifstream** և **fstream** կլասերը համապատասխանաբար ժառանգված են **ostream**, **istream** և **iostream** դասերից: Դրա շնորհիվ՝ մենք կարող ենք **fstream** օբյեկտների հետ աշխատել ծնող դասերի անդամ-ֆունկցիաներով:

Տեքստային ռեժիմում ֆայլերի հետ աշխատելիս մենք **ofstream**, **ifstream** և **fstream** կլասերի օբյեկտների հետ կաշխատենք, ինչպես **cin**-ի և **cout**-ի հետ: Ստորև բերված օրինակում մենք օգտագործում ենք << օպերատորը.



```
// grenq 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.

Ինֆորմացիայի մուտքը կազմակերպում ենք ճիշտ այնպես, ինչպես **cin** -ի հետ.

```
// kardang textayin file
#include <iostream.h>
#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 էկրանի վրա: Այստեղ մենք օգտագործեցինք մի նոր անդամ-ֆունկցիա՝ **eof**: Այս ֆունկցիան **ifstream**-ը ժառանգել է **ios** կլասից: Սա վերադարձնում է **true**, եթե հասել է 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 կարդալուց կամ գրելուց որևէ պրոբլեմ 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 ինֆորմացիայի ֆորմատավորման սխալ (օր.՝ ծրագիրը փորձում է որևէ թիվ կարդալ, իսկ ստանում է տեքստ):

**eof()**

Վերադարձնում է **true**, երբ կարդացման համար բացված ֆայլը հասել է վերջին (այլևս կարդալու սիմվոլ չկա):

**good()**

Վերադարձնում է **false** բոլոր այն դեպքերում, երբ նախորդներից որևէ մեկը կվերադարձնեք **true**:

## get և put հոսքային ցուցիչներ

Բոլոր մուտքի-ելքի օբյեկտները ունեն գոնե մեկ հոսքային ցուցիչ:

- **istream**-ը **istream**-ի նման ունի ցուցիչ, որը կոչվում է *get pointer*. այն ցույց է տալիս հաջորդ կարդացվելիք էլեմենտը:
- **ostream** -ը **ostream**-ի պես ունի ցուցիչ, որը կոչվում է *put pointer*. այն ցույց է տալիս այն տեղը, որտեղ պետք է գրվի հաջորդ էլեմենտը:
- Եվ վերջապես **fstream** -ը **iostream**-ի նման 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 *get* հոսքային ցուցիչի ներկա դիրքը (**tellg**-ի դեպքում) կամ *put* հոսքային ցուցիչի ներկա դիրքը (**tellp**-ի դեպքում):

**seekg()** և **seekp()**

Մրանք օգտագործվում են *get* և *put* հոսքային ցուցիչների դիրքը 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 ֆունկցիաները օգտագործելիս կարող ենք նշել, թե որտեղից հաշվել տրված դիրքը: *ուղղությունը* կարող է լինել հետևյալներից որևէ մեկը.

<b>ios::beg</b>	դիրքը հաշված հոսքի (ֆայլի) սկզբից
<b>ios::cur</b>	դիրքը հաշված ցուցիչի ներկա դիրքից
<b>ios::end</b>	դիրքը հաշված հոսքի վերջից

Ե՛վ *get*, և՛ *put* ցուցիչների արժեքները տեքստային ֆայլերի համար և երկուական ֆայլերի համար հաշվվում են տարբեր ձևի (դա հանդիսանում է այն բանի հետևանք, որ տեքստային ֆայլերի հետ աշխատանքի ժամանակ անտեսվում են որոշ հատուկ սիմվոլներ): Այս պատճառով խորհուրդ է տրվում օգտագործել **tellg** և **tellp** ֆունկցիաների միայն առաջին նախատիպերը: Երկուական ֆայլերի հետ կարելի է օգտագործել երկու նախատիպներն էլ:

Հաջորդ օրինակը օգտագործում է այս ֆունկցիաները, որպեսզի որոշել երկուական ֆայլի չափը:

```
// obtaining file size
#include <iostream.h>
#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 bytes e.**

## Երկուական ֆայլեր

Երկուական ֆայլերի մեջ ինֆորմացիայի մուտքը/ելքը << և >> օպերատորների ինչպես նաև **getline** ֆունկցիայի միջոցով իմաստալից չէ, սակայն դրանք լիովին թույլատրելի գործողություններ են:

Հոսքերը ունեն երկու հատուկ ֆունկցիաներ մուտքի/ելքի համար. դրանք են՝ **write** և **read**: Դրանցից առաջինը (**write**) **ostream** կլասի անդամ-ֆունկցիա է և ժառանգված է **ofstream**-ի կողմից, իսկ **read**-ը **istream** կլասի անդամ է և ժառանգված է **ifstream**-ի կողմից: **fstream** կլասի օբյեկտները ունեն և՛ **write**, և՛ **read** ֆունկցիաները: Այդ ֆունկցիաների նախատիպերը հետևյալն են:

```
write ( char * buffer, streamsize չափ );
read ( char * buffer, streamsize չափ );
```

Որտեղ *buffer* -ը հիշողության հասցե է, որտեղ գրվում է կարդացված ինֆորմացիան, և որտեղից կարդացվում է գրվելիք ինֆորմացիան: Իսկ *չափ* արգումենտը ամբողջ թիվ է, որը ներկայացնում է *buffer*-ից(ում) կարդացվելիք/գրվելիք սիմվոլների քանակը:

```
// kardum eng erkuakan file
#include <iostream.h>
#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 *buffer*-ի հետ: *buffer*-ը 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 *buffer*-ի 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 (synchronization) և տեղի է ունենում հետևյալ դեպքերում.

- **Երբ ֆայլը փակվում է.** ֆայլը փակելուց առաջ բոլոր բուֆերները սինխրոնիզացվում են:
- **Երբ բուֆերը լցվում է.** բուֆերները ունեն որոշակի չափեր, և երբ բուֆերը ամբողջովին լցվում է այն սինխրոնիզացվում է:
- **Հատուկ դեպքերում. *flush* և *endl*.** ազդակները օգտագործելիս բուֆերը սինխրոնիզացվում է:
- ***sync()* ֆունկցիայի կանչի դեպքում. *sync()*** (արգումենտներ չկան) անդամ-ֆունկցիայի կանչը բերում է անմիջապես սինխրոնիզացիայի: Այս ֆունկցիան վերադարձնում է **int** տիպի արժեք, որը հավասար է (-1)-ի, եթե հոսքին ոչ մի բուֆեր կապած չէ կամ եթե որևէ պրոբլեմ է առաջացել: