ԾՐԱԳՐԱՎՈՐՄԱՆ C++ ԼԵԶՈՒ

Ծրագարավորման ցանկացած լեզու հնարավորություն է ընձեռնում մեքենային հասկանալի տեսքով նկարագրելու ալգորիթմներ:

Ալգորիթմը իրագործողին ուղղված, նրան հասկանալի այնպիսի գործողությունների հաջորդականություն է, որոնց կատարումը բերում է առաջադրված խնդրի լուծմանը:

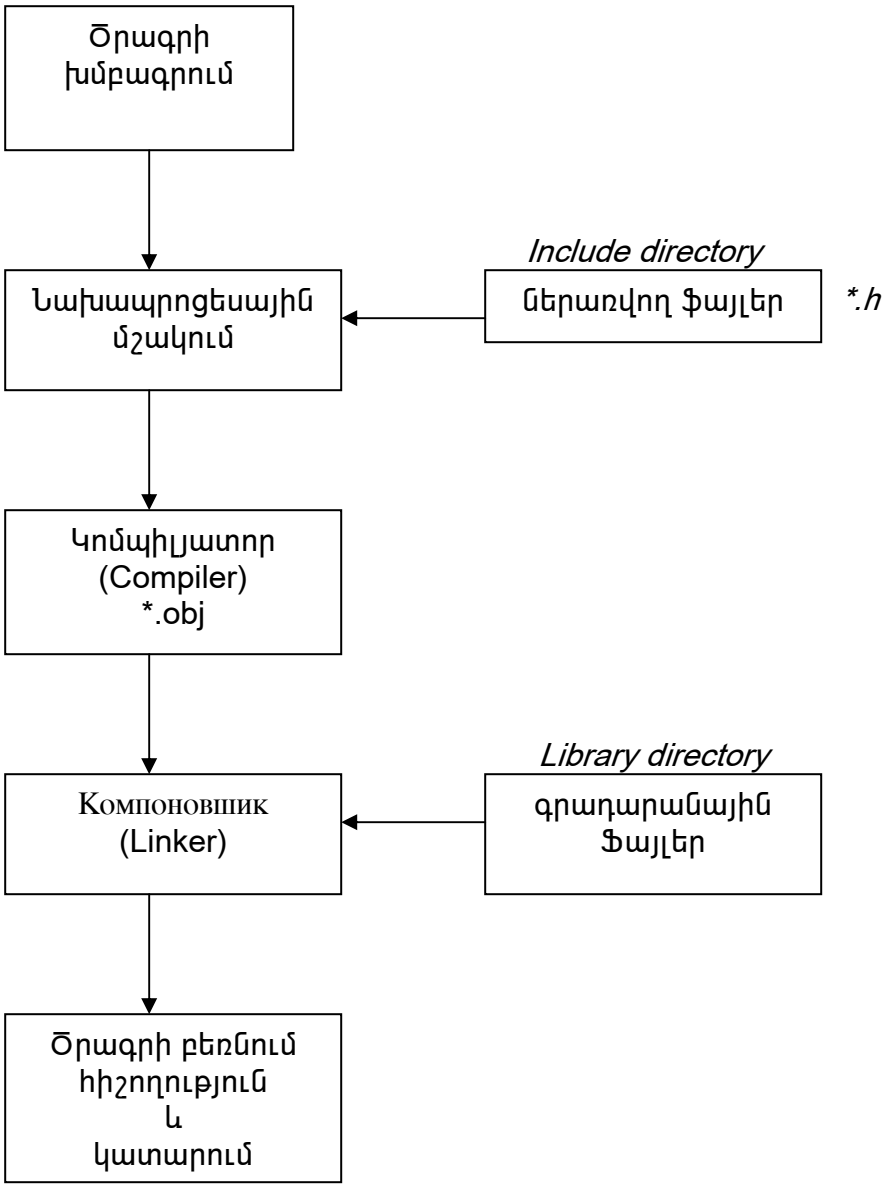
Ալգորիթմի հիմնական հատկություններն են.

- Վերջավորություն - ալգորիթմն պետք է վերջավոր քայլերից հետո ավարտի իր աշխատանքը
- Միարժեքություն - ալգորիթմի յուրաքանչյուր քայլից հետո հաջորդ քայլի ընտրությունը կատարվում է միարժեք ձևով
- Ընդհանրություն կամ մասայականություն - ալգորիթմն պետք է նախատեսված լինի լուծելու ոչ թե 1 որոշակի խնդիր, այլ նույնատիպ խնդիրների մի ամբողջ դաս
- Մեխանիկորեն կատարելու հնարավորություն - ալգորիթմ ստեղծելը պահանջում է որոշակի գիտելիքներ և հմտություններ այս կամ այն բնագավառից, սակայն երբ ալգորիթմն արդեն ստեղծված է, նրա ծրագրավորումը ոչ մի գիտելիքներ չի պահանջում, այլ պահանջում է միայն ալգորիթմով առաջադրված քայլերի կատարում, որը և կատարում է հաշվիչ մեքենան:

C++ լեզվով ծրագիր գրելիս կատարվում են հետևյալ քայլերը

- *տեքստի խմբագրում*. այն կատարվում է հատուկ խմբագրական ծրագրերում: Ծրագիրը հիշվում է հիշողությունում .c, .cpp, .cxx ընդլայնումներով
- *Կոմպիլացիա*. այս փուլը սկսվում է նախնական մշակումով, որի ժամանակ պրոցեսորային դիրեկտիվների միջոցով տեքստային ֆայլին է ավելացվում ծրագրային կտորներ, որոնք նշված են համապատասխան դիրեկտիվներում: Ծրագիրը ստուգվում է սխալներից, սխալ հայտնաբերելիս տրվում է հաղորդագրություն: Սխալները վերացնելուց հետո, ծրագիրը թարգմանվում է մեքենայական կոդի(օբյեկտային կոդ) և ստեղծվում է *.obj օբյեկտային ֆայլ:
- *Компановка.* օբյեկտային ֆայլին է ավելացվում անհրաժեշտ գրադարանային ֆունկցիաներ կամ այլ ֆունկցիաներ որոնք գտնվում են ոչ այս ծրագրում:
- *Բեռնավորում*. Ծրագրը կատարումից առաջ բեռնվում է հիշողություն:
- *Կատարում*. պրոցեսորը հաջորդաբար կատարում է ծրագրի հրամանները;

Ծրագրի աշխատանքի սխեմատիկ պատկերը:



C++ լեզվի հիմնական տիպերը հետևյալն են

- Տրամաբանական
- Նշանային
- Ամբողջ
- Իրական
- դատարկ (void)

Տրամաբանական տիպ (bool)

Տրամաբանական տիպի փոփոխականերն կարող են ընդունել ***true*** կամ ***false*** արժեքները: ***false***-ին համապատասխանում է 0, իսկ ***true***-ին կամայական ոչ 0-ական ամբողջ թիվ(0,1,2…)

օրինակ՝

```
bool a; // a-ն  bool տիպի է
bool b = true; // b-ն bool տիպի է և ընդունել է true սկզբնական արժեք
a = true;
bool c = a+b // a+b = 2, ուստի c –ի արժեքը նույպես true է
```

Նշանային տիպ (char)

Այս տիպի փոփոխականի արժեք կարող են հանդիսանալ սիմվոլներից որևէ մեկը օր.՝

```
char c = ‘a’;
```

Նշանային տիպի փոփոխականի համար հիշողությունում առանձնացվում(հատկացվում) է 1 բայթ: Ամեն մի սիմվոլ ունի իր թվային անալոգը:

ծրագիր որը տալիս է սիմվոլների կոդը

```
#include <iostream.h>
void main()
{
        char c;
        cout << “input char” ;
        cin >> c;
        cout << “char” << c << “is equal ” << (int)c << endl;
}
```

Կան հատուկ սիմվոլներ որոնք նախատեսված են որոշակի գործողություններ կատարելու համար

	կոդը	
\a	0x07	ձայնային ազդանշան
\b	0x08	մի քայլ ետ
\f	0x0C	էջը փոխել
\n	0x0A	տողը փոխել
\r	0x0D	Возврат каретки
\t	0x09	հորիզոնական табулация
\v	0x0B	ուղղահայաց табулация
\\	0x5C	\ նշանը
\’	0x27	ապաթարթ
\”	0x22	չակերտ
\?	0x3F	? նշան
\000	000	սիմվոլների 8-ական կոդ
\xhh	0xhh	սիմվոլների 16-ական կոդ

Ամբողջ տիպ (int)

Կախված թե հիշողության մեջ որքան բայթ է հատկացվում int տիպը ունի հետևյալ տարատեսակները

short int // կամ short,

հիմնականում հատկացվում է 2 բայթ(կախված մեքենայի կառուցվածքից և կոմպիլյատորից), ընդունվող արժեքների տիրույթը –32768…32767

unsigned int // առանց նշանի

հատկացվում է 2 բայթ, ընդունվող արժեքների տիրույթը 0…65535

int

հատկացվում է 2 բայթ, ընդունվող արժեքների տիրույթը –32768…32767

unsigned long int // կամ unsigned long

հատկացվում է 4 բայթ, ընդունվող արժեքների տիրույթը 0…4294967295

long

հատկացվում է 4 բայթ, ընդունվող արժեքների տիրույթը -2147483648…2147483647

մասնակի թվերը գրվում են 10-ական համակարգով

օր.՝ 1, 63, 11254, 85

8-ական համակարգով գրելիս, սկզբում ավելացնում ենք 0

օր.՝ 02, 077, 0125

16-ական համակարգի դեպքում ավելացնում ենք 0x

օր.՝ 0x25, 0xF4, 0x45, 0x3D

sizeof() ֆունկցիայի միջոցով կարելի է իմանալ տվյալ տիպին հատկացվող բայթերի քանակը

*cout << "\nsize of int" << **sizeof(int)**;* // կտպի int հատկացվող բայթերի քանակը

*cout << "\nsize of unsigned int" << **sizeof(unsigned int)**;* // կտպի unsigned int հատկացվող բայթերի քանակը

*cout << "\nsize of long" << **sizeof(long**);* // կտպի long հատկացվող բայթերի քանակը

*cout << "\nsize of short" << **sizeof(short)**;* // կտպի short հատկացվող բայթերի քանակը

Ամբողջ տիպի հաստատուներ նկարագրելու ժամանակ, երբ ծրագրավորողին հարմար չէ այն տիպը որը հատկացվում է կոմպիլյատորի կողմից, նա կարող է բացահայտ ձևով ազդել՝ ավելացնելով L, l(long) կամ U, u(unsigned)տառերը:

օր.՝ 64L –ի դեպքում կհատկացվի 4 բայթ, քեպետև 64-ը կարող էր լիներ int (2 բայթ);

*cout<< "\nsize of 45 " << **sizeof** 45*; // կտպի 2
*cout<< "\nsize of 45u " << **sizeof** 45u;* // կպտի 2
*cout<< "\nsize of 45L " << **sizeof** 45L;* // կտպի 4
*cout<< "\nsize of 45LU " << **sizeof** 45LU;* // կպտի 4

այստեղ օգտագործվել է унарная sizeof գործողությունը, որը ցույց է տալիս աջ կողում գրված օպերանդի հիշողությունում հատկացվող բայթերի քանակը:

Իրական տիպ(float, double)

իրական տիպ ունի հետևյալ տարատեսակները

float
հատկացվում է 4 բայթ, ընդունվող արժեքների տիրույթը 3.4E-38…3.4E+38

$(3.4*10^{-38} ..3.4*10^{38})$

double
հատկացվում է 8 բայթ, ընդունվող արժեքների տիրույթը 1.7E-308…1.7E+308

long double
հատկացվում է 10 բայթ, ընդունվող արժեքների տիրույթը 3.4E-4932…1.1E+4932

*cout << "\nsize of float" << **sizeof (float)**;* // կտպի float հատկացվող բայթերի քանակը

*cout << "\nsize of double" << **sizeof (double)**;*//կտպի double հատկացվող բայթերի քանակը

*cout << "\nsize of long double" << **sizeof (long double);*** // կտպի long double հատկացվող բայթերի քանակը

Դատարկ տիպ (void)

void տիպի օբյեկտ գոյություն չունի: Սահմանվում է հիմնականում այն դեպքերում, երբ ցուցիչի տիպը դեռևս պարզ չի և կարող է փոփոխվել: Օգտագործվում է նաև ֆունկցիաների նկարագրման ժամանակ:

օր.՝

void p*; // p հետագայում կարող է տալ և int, և float, և մյուս տիպերը

***int** a; **float** b;*

p = &a; p = &b;

void f() //ֆունկցիան ոչ մի արժեք չի վերադարձնում

Բացի ստանդարտ տիպերից, ծրագրավորողը ինքը կարող է նկարագրել(ստեղծել) նոր տիպեր: Դրանցից են թվարկումային տիպը և ստրուկտուրաները(class, struct, union):

Գոյություն ունեն նաև ցուցչային, հղումային տիպերը ինչպես նաև զանգվածներ:

Թվարկումային տիպ (enum)

Թվարկումային տիպը դա ամբողջ տիպի հաստատուներ են, որոնք ունեն ունիկալ անուներ:

օր.՝

enum *{one = 1, two = 2, tree = 3};*

one, two, tree-ն դրանք պայմանական անուներ են, որոնք տալիս է ծրագրավորողը և ծրագրի մեջ կարող ենք այդ թվերի փոխարեն օգտագործել այդ անունները:

մեզ ծանոթ bool տիպը սահմանվել է այսպես

enum *bool{false = 0, true = 1};*

եթե թվարկումային տիպի գրելու ժամանակ = նշանը բաց է թողնված, ապա համարակալումը կկատարվի ավտոմատ կերպով և համարակալումը սկսվում է 0-ից:

օր.՝

enum *week{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}*
0 1 2 3 4 5 6

եթե թվարկումային տիպի ժամանակ մի անդամի համար բացահայտ նշված է արժեքը, իսկ մյուս անդամներինը բաց է թողնված, ապա բաց թողնվածների համար համարակալումը կատարվում է ավտոմատ՝ ավելացնելով 1-ով:

օր.՝

enum color{red = 3, green, blue}
4 5

Ցուցիչ(Указатель)

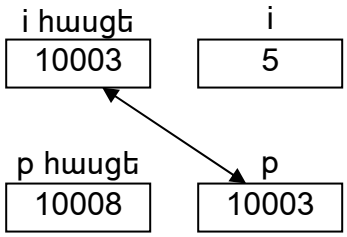
Ցուցչային տիպի օբյեկտը իր մեջ պարունակում է մեկ այլ օբյեկտի հասցե: Հիմնականում այն օգտագործում են ֆունկցիաներին՝ որպես պարամետր, մեծ օբյեկտներ փոխանցելու դեպքերում: Սահմանվում են հետևյալ կերպ՝

տիպ *ցուցչի_անուն

օր.՝

```
int *p; // p-ն ցուցիչ է, որը ցույց է տալիս int տիպի փոփոխականի
```

```
int i = 5;
int *p = &i;
```



* օպերանդ գործողությունը կոչվում է անուղակի հասեավորում(косвенное адресация)

& օպերանդ գործողությունը տալիս է օպերանդի հասցեն

*p կտա 5, անուղակի ձևով դիմում ենք i-ին, և կարող ենք փոխենք i –պարունակությունը,

*p = 7; // i-ն անուղղակի կդառնա 7

օր.՝

```
#include <iostream.h>

void main()
{
        int *p1, *p2; // p1, p2 int տիպի ցուցիչներ են
        int c = 5, d = 3;
        double k = 9;
        double *pd = &k; // pd double տիպի ցուցիչ է, և ցույց է տալիս k-ին
        p1 = &c; // p1 ցույց է տալիս c-ին
        cout << p1 << " " << *p1 << endl; //  տպում է c-ի հասցեն, եվ c-ի արժեքը
        *p1 = 10; // դա նույնն է c=10
        cout << "\nc = " << c << endl; //տպում է c
        p1 = &d; // p1 ցույց է տալիս d
        p2 = p1;
        cout << p2 << " " << *p2 << endl; // տպում է d-ի հասցեն և արժեքը
        void* pv;
        pv = p1;
        pv = pd;
}
```

Հղումային տիպ

Հղումային տիպի փոփոխականը հանդիսանում է մեկ այլ օբյեկտի «հոմանիշ» և նկարագրելիս պարտադիր պետք է տրվի սկզբնական արժեք: Նկարագրվում է հետևյալ կերպ

տիպ &անուն = սկզբնական արժեք;

Հղումը, ինչպես նաև ցուցիչը անուղակի փոխում են այն փոփոխականների արժեքը, որոնց վրա նրանք «հղված» են:

Հղումը հիմնականում օգտագործվում է ֆունկցիաներում, որպես ֆորմալ պարամետր:

Հղման ժամանակ փոփոխականների հետ կապվում է երկու անուն՝ ինքը և իր հղումը:

օր.՝

```
int a = 10;
int &ra = a;
```

եթե ra = 3, ապա a-ն նույպես կդառնա 3

Ի տարբերություն ցուցչային տիպի, հղումային տիպի փոփոխականը նկարագրությունից հետո չի կարող «հղվել» մեկ այլ փոփոխականի:

օր.՝

```
int a = 3;
int b = 4;
int &ra = a;
&ra = b; // սխալ է
ra = b;// դա նույնն է a = b , այսինքն 4
```

Զանգված(array)

Զանգվածը միևնույն տիպի էլեմենտների հավաքածու է, որի անդամներին կարելի է դիմել ինդեքսի միջոցով: Սահմնանվում է հետևյալ կերպ

տիպ անուն[չափ]

օր.՝

```
int mas[10];
```

սահմանվել է mas անունով 10 հատ ամբողջ էլեմենտներով զանգված:

Ինդեքսի համարակալումը սկսվում է 0-ից, մեր օրինակի համար ինդեքսը փոխվում է 0-ից մինչև 9-ը: Զանգվածի չափը պետք է լինի հաստատուն արտահայտություն (const):

օր.՝

```
const int size = 10;
char name[size];
```

սահմանվեց 10 չափանի name զանգված, որի էլեմենտները char տիպի են:

Զանգվածը սահմանելիս կարելի է բացահայտ կերպով տալ անդամների սկզբնական արժեքներ, որը կատարվում է հետևյալ կերպ

օր.՝

```
int a[4] = {3, 7, 5, 4}; // a[0] = 3, a[1] = 7, a[2] = 5, a[3] = 4
```

այս օրինակում նշեցինք չափը, որը պարտադիր չէ, եթե տալիս ենք զանգվածի անդամների սկզբնական արժեքները

օր.՝

```
int b[] = {9, 5, 6, 8, 4}; // սահմանվել է 5 չափանի մասիվ
```

Սիմվոլային զանգվածի սկզբնական արժեքը կարելի է տալ ոչ միայն {}-ի միջոցով, այլև տողային տեսքով;

օր.՝

```
char name[7] = {‘A’, ‘r’, ‘m’,  ‘e’, ‘ n’,  ‘i’,  ‘a’};
char nema1[] = “Armenia”;
```

տարբերությունը կայանում է նրանում, որ տողային ձևը վերջանում է ՛0՛ -ական սիմվոլով (որը ցույց է տալիս տողի վերջը), այդ պատճառով name1 –ի չափը կլինի 8, այլ ոչ է 7:

Զանգվածին՝ որպես սկզբնական արեք չի կարելի տալ մեկ ուրիշ զանգված:

օր.՝

```
int mas1[] = {1, 2, 3, 4, 10};
int mas2[] = mas1; // սա սխալ է;
```

Բազմաչափ զանգվածը սահմանվում է հետևյալ կերպ՝

օր.՝

```
int a[4][3] = { {0, 1 ,2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}};
```

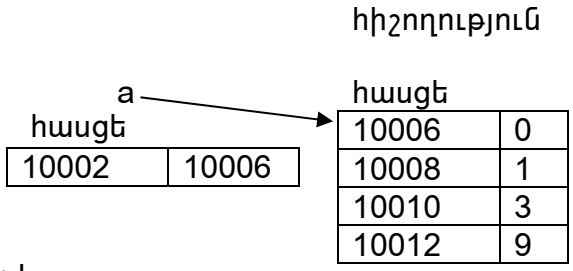
կամ

```
int b[4][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

Զանգվածի անունը, իրենից ներկայացնում է իր առաջին էլեմենտի հասցեն, այստեղից զանգվածի առաջին էլեմենտին կարելի է դիմել երկու ձևով

օր.՝

```
short a[] = {0, 1, 3, 9};
```



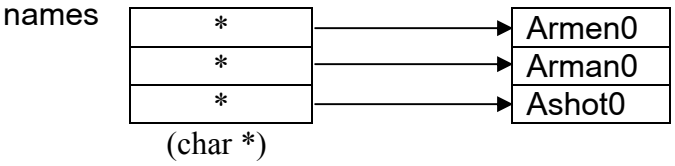
a –ն ինքը հասց է առաջին էլեմենտին a[0] կամ *a

```
int *p; // p-ն int տիպի ցուցիչ է
p = a; //p-ն ցույց է տալիս a զանգվածի առաջին էլեմենտը
p++; // p-ն ցույց է տալիս a զանգվածի երկրորդ էլեմենտը
cout << *p; // կտպի 1, երկրորդ էլեմենտի արժեքը
```

զանգվածի 5-րդ էլեմենտին դիմելու համար կարելի է գրել *(a+5), նունն է ինչ որ a[5]

Սիմվոլների զանգվածը անվանում են նաև C տող: այդ տիպի հետ աշխատելիս ավելի հեշտ է, երբ նկարագրում ենք այսպես

```
char  *string = “Russia”;
char* names[] = {“Armen”, “Arman”, “Ashot”};
```



Հաստատուներ(const)

Շատ օբյեկտներ ինիցիալիզացիայից հետո չեն փոփոխվում: Այդպիսի փոփոխականները (օբյեկտներ) նկարագրվում են որպես հաստատուն:

օր.՝

```
const int model = 90;
const int v[] = {1, 2, 3, 4};
void g(const int * p)
{
        // ծրագրի ներսում չի կարելի փոփոխել *p
}
```

Ցուցիչը և Հաստատունը

Գոյություն ունեն հետևյալ տարատեսակները հաստատուն ցուցիչ, ցուցիչ հաստատունի վրա, հաստաուն ցուցիչ հաստատունի վրա:

օր.՝

```
char s[] = “Corn”;
char p[] = “Titanik”;
char* const cp = s; // հաստատուն ցուցիչ
char const * pc; // ցուցիչ հաստատուի վրա const char* pc;


cp[3] = ‘k’; // ճիշտ է,
cp = p; // սխալ է, քանի որ cp –ն հաստատուն ցուցիչ է և չի կարող
          //փոփոխվել
pc = s;
```

```
pc[3] = 'b'; // սխալ է, քանի որ pc –ն ցուցիչ է հաստատուի վրա, և այդ
             // հաստատունը չի կարելի փոփոխել
pc = p; // ճիշտ է
const char* const cpc = s; // հաստաուն ցուցիչ հաստատուի վրա
```

typedef –ի նկարագրությունը

typedef –ի միջոցով տիպերին կարելի է տալ նոր անուն: typedef –ի միջոցով նոր տիպ չի ստեղծվում, այլ ուղակի նրան տրվում է մի նոր անուն(հոմանիշ)՝ ավելի կարճ, օգտագործելու համար հարմար լինելու համար:

օր.՝

```
typedef unsigned char UC; // unsigned char-ի  փոխարեն կարող ենք օգտագործել UC
typedef unsigned long int int32; // unsigned long int → int32
typedef float* PF;
```

ծրագրում օգտագործվում է հետևյալ կերպ՝

```
UC c; // c-ն  unsigned char տիպի է
PF p; // p-ն float տիպի ցուցիչ է
int32 i; // i-ն unsigned long int տիպի է
```

Փոփոխականներ և նրաց նկարագրությունը

Փոփոխականը(օբյեկտը) իրենից ներկայացնում են հիշողության մեջ հատկացված որոշակի տեղ, որը ունի անուն: Փոփոխականները պարտադիր պետք է նկարագրվեն՝ նշվեն թե ինչ տիպի են: Փոփոխականի տիպից կախված հիշողությունում հատկացվում է համապատասխան քանակով բայթեր, որոնցում էլ պահվում է փոփոխականի արժեքը:

Փոփոխականի հետ կապված է 2 մեծություն՝ հասցե և արժեք:

- արժեք(r արժեք), որը պահպանվում է հիշողությունում
- հասցե(l արժեք), որտեղ պահպանվում է r արժեքը

փոփոխականի հետ կապված է նաև նրա տիպը, հիշողության դասը և տեսանելիության տիրույթը:

Փոփոխականները նկարագրվում են հետևյալ կերպ

S m տիպ անուն1 սկզբ. արժեք1, անուն2 սկզբ. արժեք2….

S –ը հիշողության դաս, որը կարող է լինել ***auto***, ***static***, ***extern***, ***register***, որոնցով պայմանավորված է, թե ինչպես է այդ օբյեկտը հիշողությունում տեղադրվում և ինչքան ժամանակ:

auto -այս փոփոխականներին հիշողությունը տրվում է բլոկ մտնելիս և ազատվում է, երբ դուրս ենք գալիս բլոկից

register-նման է ***auto***-ին ի տարբերություն, որ նա օգտագործում է ռեգիստորները: երբ ռեգիստորները զբաղված են լինում ուրիշ օբյեկտներով, ապա այդ փոփոխականը դիտվում է որպես ***auto***

static - այս փոփոխականները գոյություն են ունենում այն ֆայլի մեջ որտեղ նրանք նկարագրված են

extern -գլոբալ օբյեկտներ են, որոնց կարելի է դիմել ծրագրային մյուս մոդուլներից

m –ը մոդիֆիկատոր է, որը կարող է լինել ***const*** կամ ***volatile***

սկզբնական արժեքը կարելի է տալ 2 ձևով

օր.՝

int *a = 7*; կամ ***int*** *a(7);*

Տիպերի վերափոխում(преобразование типов)

Տիպերի բացահայտ վերափոխում(ձևախոխում) գործողությունը ունի երկու տարատեսակ՝ կանոնական և ֆունկցիոնալ:

Կանոնականը ունի հետևյալ տեսքը՝

(տիպ) օպերանդ

օպերանդի արժեքը վերափոխում(ձևախոխում) է փակագծերում գրված տիպի: Եթե իրական տիպի արժեքը ձևախոխում է ամբողջ տիպի, ապա հնարավոր է սխալներ կապված ամբողջ թվի ներկայացման թույլատրվող տիրույթի հետ:

օր.՝

float *a = 3.5;*

*(**int**) a*; // այս արտահայտության արժեքը կլինի 3

կանոնական ձևում թույլատրելի է նաև հետևյալ տեսքի արտահայտությունները, որոնք չի կարելի գրել ֆունկցիոնալ դեպքում:

օր.՝ *(**unsigned long**)(x / 3 + 2);*

Ֆունկցիոնալ ձևը ունի հետևյալ տեսքը՝

տիպ (օպերանդ)

օր.՝

***int**(3.14), **float**(2 / 5), **int**('A')*

Որոշակի գործողություններ կատարելիս օպերանդները պետք է ունենան համապատասխան տիպ, իսկ եթե այդ տիպի չեն, ապա «ստիպողաբար» կատարվում է տիպերի ձևափոխում:

օր.՝

```
float a = 3.4;
int b = a + 1.3; // b կընդունի 4
int c = 5;
a = b / c; // a կընդունի 1, քանի որ b և c ամբողջ տիպի են, բաժանումը
           //կատարվել է այդ տիպի համար => ստացվել է 1
```

ճիշտ արդյունք ստանալու համար պետք է գրել

*a = (**float**)b / c; // կամ a = b / (**float**)c;*

Ստորև բերված է այն ձևափոխությունները, որոնց դեպքում ճշտությունը և թվային արժեքը պահպանվում է

***signed char** → **short** → **int** → **long** ;*
***unsigned char** → **unsigned short** → **unsigned int** → **unsigned long**;*
float** → **double** → **long double

Գործողության նշանները

***օպերանդ**

գործողությունը կոչվում է անուղակի հասեավորում(косвенное адресация), և տալիս է հիշողությունում օպերանդ հասցեում գրված արժեքը(օպերանդը պետք է լինի հասցե)

& օպերանդ

գործողությունը տալիս է օպերանդի հասցեն

```
&a; // տալիս է a-ի հասցեն
```

Տրամաբանական գործողություններ

տրամաբանական ժխտում

! օպերանդ

օր.՝

!1 = 0; !0 = 1; !3 = 0; !true = false; !false = true; !(-5) = 0;

տրամաբանական «կամ»

օպերանդ1 || օպերանդ2

օր.՝

true || true = true; 1 || 1 = 1;

true || false = true; 1 || 0 = 1;

false || true = true; 0 || 1 = 1;

false || false = false; 0 || 0 = 0;

տրամաբանական «և»

օպերանդ1 && օպերանդ2

օր.՝

true && true = true; 1 && 1 = 1;

true && false = false;

false && true = false;

false && false = false;

<, > փոքր է, մեծ է

<=, >= փոքր կամ հավասար է, մեծ կամ հավասար է

== հավասար է

!= անհավասար է

Մորգանի օրենք.

!(a && b) = (!a || !b) և (!a || !b) =!(a && b)

Թվաբանական գործողություններ

++ - մեկով մեծացում

ունի երկու տարատեսակ префикс և постфикс

префикс - ++a; // մեծացնում է a-ի արժեքը 1-ով, և օգտագործվում է ավելացված արժեքը

постфикс – a++;//օգտագործում է a-ի արժեքը, որից հետո մեծացնում է նրա արժեքը 1-ով

օր.՝

```
int a(4);
cout << ++a; // a + 1, և կտպի 5
cout << a;     // կտպի 5
cout << a++; // կտպի 5, որից հետո a + 1
cout << a;    // կտպի 6
```

-- մեկով նվազում

ունի երկու տարատեսակ префикс և постфикс

префикс - --a; // նվազեցնում է a-ի արժեքը 1-ով, և օգտագործվում է փոփոխված արժեքը

постфикс - a--;//օգտագործում է a-ի արժեքը, որից հետո նվազեցնում է նրա արժեքը 1-ով

օր.՝

```
int b(5);
cout << --b; // a - 1, և կտպի 4
cout << b;     // կտպի 4
cout << b--; // կտպի 4, որից հետո b - 1
cout << b;    // կտպի 3
```

+= `a+= 5;` // համարժեք է a = a +5;

*= `a *= 5;` // համարժեք է a = a * 5;

-= `a -= 5;` // համարժեք է a = a - 5;

օր.՝

```
a += 3 – d; // a = a + 3 – d;
a -= 3 – d; // a = a – (3 - d) = a – 3 + d;
p *= 3 – b; // p = p * (3 – b);
c /= 5;       // c = c / 5;
```

% -ամբողջ թվերի բաժանամաև դեպքում տալիս է մնացորդը

օր.՝

```
5 % 2 = 1;  6 % 4 = 2;
```

Տեղաշարժի գործողություններ

Որոշված է միայն ամբողջ թվերի համար

ձախ օպերանդ << աջ օպերանդ

ձախ օպերանդի բիթային ներկայացված արժեքը աջ օպերանդի արժեքի չափով տեղաշարժում է ձախ

ձախ օպերանդ >> աջ օպերանդ

ձախ օպերանդի բիթային ներկայացված արժեքը աջ օպերանդի արժեքի չափով տեղաշարժում է աջ

օր.՝

4 << 2 = 16;

$4 = (100)_2$; $(100)_2$ << 2 = $(10000)_2 = (16)_{10}$

9 >> 3 = 1;

$9 = (1001)_2$; $(1001)_2$ >> 3 = $(1)_2 = 1$;

Բիթային գործողություններ

& կատարում է բիթային «և» գործողությունը

օր.՝

13 & 7 = 5;

```
  1101
& 111
------
  101
```

| կատարում է բիթային «կամ» գործողությունը

օր.՝

4 | 2 = 6;

```
  100
| 10
-----
  110
```

^ կատարում է բիթային «Ժխտող կամ» գործողությունը

1 ^ 1 = 0; 0 ^ 0 = 0;
1 ^ 0 = 1; 0 ^ 1 = 1;

օր.՝

6 ^ 5 = 3;

վարժ. գրել արտահայտությունների արժեքները

int k = 35 / 4; k /= 1 + 1 + 2; k *= 5 – 2; k %= 3 + 2; k += 21 / 3;

k -= 6 – 6/2; k <<= 2; k >>= 6-5; k &= 9 + 4; k |= 8 –2; k ^= 10;

new և delete գործողությունները

Այս գործողությունները նախատեսված են դինամիկ հիշողություն բախշելու համար: Ունի հետևյալ տեսքը՝

new տիպ կամ new տիպ սկզբնական_արժեք

հիշողության ազատ մասում հատկացվում է տիպին համապատասխան քանակով հիշողություն, և եթե կա սկզբանական_արժեք ապա այդ արժեքը գրվում է հատկացված հիշողությունում: new գործողության հաջող կատարման դեպքում վերադարձվում է հատկացված հիշողության հասցեն, հակառակ դեպքում(հնարավոր է որ ազատ հիշողություն չլինի) վերադարձնում է null: Հատկացված հիշողության հասցեն հասանելի է դառնում ցուցիչների կիրառման միջոցով, որը կատարվում է հետևյալ կերպ՝

ցուցիչ = new տիպ սկզբնական_արժեք

նշենք, որ ցուցիչը պետք է լինի նույն տիպի:

օր.՝

```
float* fp = new float (3.14); //
```

new float (3.14) –ի միջոցով հատկացվում է 4 բայթ, որտեղ գրվում է 3.14, իսկ fp ցուցիչը ցույց է տալիս այդ հասցեն:

Մասիվների դեպքում նունպես կարելի է դինամիկ հիշողություն հատկացնել, որը կատարվում է հետևյալ կերպ:

տիպ *ցուցիչ = new տիպ [չափ];

օր.՝ ստեղծենք 10 չափանի int տիպի մասիվ mas անունով

```
int *mas = new int[10];
```

Բաղմաչափ զանգվածների դեպքում՝

```
const int n = 5;
const int m = 8;
int mas[n][m];
```

դինամիկ հիշողության հատկացման դեպքում՝

```
cin>>n>>m;
int **mas = new int*[n];
for(int i = 0; i<n; i++)
        mas[i] = new int[m];
```

կամ

```
int **mas = (int**) new int[n][m];
```

Բացահայտ կերպով հիշողությունն ազատելու համար օգտագործում են delete գործողությունը, որը ունի հետևյալ տեսքը՝

delete ցուցիչ;

որտեղ ցուցիչը new գործողությունով հատկացված հիշողության հասցե է, իսկ մասիվների դեպքում delete ունի հետևյալ տեսքը՝

delete [ ] ցուցիչ:

օր.՝

delete [ ] mas; delete fp;

Գործողությունների կատարման հաջորդականությունը

ռանգը	գործողությունը	ասոցիատիվությունը
1	() [] -> :: .	→
2	delete, new, sizeof, *, &, --, ++, -, +, ~, !	→
3	.*, ->*	→
4	*, /, %	→
5	+, -	→
6	<<, >>	→
7	<, <=, >=, >	→
8	==, !=	→
9	&	→
10	^	→
11	|	→
12	&&	→
13	||	→
14	?:	→
15	=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=	←

Հրահանգներ

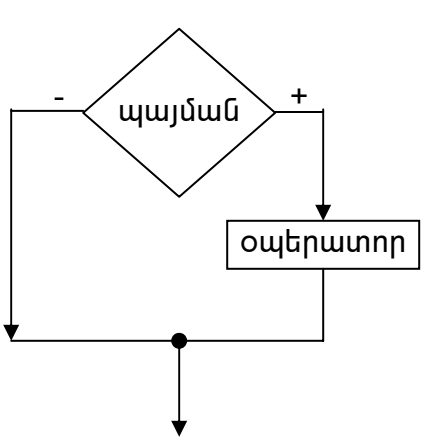
Պայմանով հրահանգ(*if*).

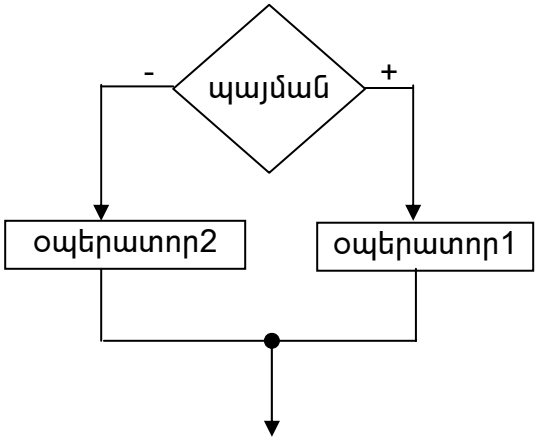
ունի հետևյալ կառուցվածքը

```
if(պայման)
        օպերատոր
```

պայմանը պետք է լինի bool տիպի արտահայտություն: Պայմանը true լինելու դեպում(ոչ 0-ական) կատարվում է օպերատորը: Այս տեսքը կոչվում է թերի տեսք, իսկ լրիվ տեսքը ունի հետքյալ կառուցվածքը՝

```
if(պայման)
        օպերատոր1
else
        օպերատոր2
```





օր.՝

```
int max;
if(a > b)
        max = a;
else
        max = b;
```

երե պայմանի ճիշտ կամ սխալ լինելու դեպքում պետք է կատարել մի քանի գործողություն, ապա այդ գործողությունները ներառում ենք մի բլոկի մեջ:

օր.՝

```
if( a > b)
{
        max = a;
        cout << “max = ” << a ;
}
```

գոյություն ունի պայմանով հրահանգի գրառման մի ուիշ տեսակ

պայման ? արժեք1 : արժեք2

պայմանի true լինելու դեպքում վերադարձնում է արժեք1-ը, իսկ false-ի դեպքում արժեք2-ը օր.՝

```
max = a > b ? a : b;
```

Պայմանի հրահանգ(switch)

ունի հետևյալ տեսքը

```
switch(արտահայտություն)
{
        case հաստատուն արտ1 : օպերատոր1;
        case հաստատուն արտ2 : օպերատոր2 ;
        ….
        default:  օպերատոր;
}
```

switch օպերատորը ղեկավարությունը տալիս է այն օպերատորին, որի case-ի արտահայտությունը համընկնում է switch-ի արտահայտության հետ, եթե ոչ մի համընկնում չկա ապա ղեկավարությունը տրվում է default-ի օպերատորին: case-ի մեջ օգտագործվում է break օպերատորը, որպեսզի տվյալ case-ի օպերատորը(կամ օպերատորները) կատարելուց հետո չկատարի հաջորդող case-երի օպերատորները

օր.՝

```
int k;
cin >> k;
switch(k)
{
        case 1 : cout << “You have pressed 1\n”;
                 break;
        case 2:  cout << “You have pressed 2\n”;
                 break;
        //….
        default : cout << “You have pressed the chars\n”;

}
```

Պայմանով ցիկլ*(while)*

ունի հետևյալ կառուցվածքը՝

```
while(պայման)
```

հրահանգ

քանի դեռ պայմանը ճիշտ է կատարվում է հրահանգը, եթե հրահանգները շատ են վերցվում են {}-ի մեջ:

օր.՝

```
i = 1;
while(i <= 10)
{
        cout << i * i << “\n”;
        i++;
}
```

կտպի 1-ից մինչև 10 թվերի քառակուսիները, այս նույն ծրագիրը կարելի է գրել նաև

```
i = 1;
while(i++ <= 10)
        cout << i * i << “\n”;
```

պայմանով ցիկլը ունի նաև հետևյալ տարատեսակը

```
do
   հրահանգ
while(պայման)
```

հրահանգը կատարում է այնքան մինչև պայմանը դառնում է false: Նախկին while-ից տարբերվում է նրանով, որ այս դեպքում հրահանգը առնվազն 1 -անգամ կատարվում է:

օր.՝

```
do
  a ++
while(a < 10)
```

Պայմանով ցիկլ*(for)*

ունի հետևյալ տեսքը

for(արտահայտություն1; արտահայտություն2; արտահայտություն3)

օպերատոր

արտահայտություն1 – ցիկլը ղեկավարող պարամետրին տալիս է սկզբնական արժեք,

արտահայտություն2 – հանդիսանում է ցիկլի տևողության պայմանը

արտահայտություն3 - բնորոշում է ղեկավարող պարամաետրի փոփոխման չափը

եթե արտահայտություն1..3 –ը պարունակում են շատ պարամետրերը(մեկից ավել), ապա իրարից բաժանվում են ստորակետով ՛,՛

for ցիկլին համարժեք while-ն ունի հետևյալ տեսքը

```
արտահայտություն1
while(արտահայտություն2)
{
        օպերատոր
        արտահայտություն3
}
```

օր.՝

```
for(int i = 1; i <= 100; i++)
        cout << I << “, ” ;     // կտպի 1-100 թվերը
for(int i = 2; i <= 10; i++)
        cout << I << “, ” ;     // կտպի 2, 4, 6, 8, 10 թվերը
```

break և continue օպերատորները

break և continue օպերատորները փոխում են ղեկավարումը: Երբ break օպերատորը կատարվում է while, for, do/while ցիկլերում կամ switch-ում, ապա անհապաղ դուրս է գալիս այդ ցիկլերից և ծրագիրը կատարում է ցիկլիկ օպերատորից հետո գրված հաջորդ հրամանը: continue օպերատորը while, for, do/while ցիկլերի մարմիներում բաց է թողնում իրենից հետո գրված օպերատրեների աշխատանքը և ղեկավարությունը տալիս է ցիկլի հաջորդ իտերացիային:

օր.՝

```
for(int i = 1; i <= 10; i++)
{
        if( i == 5)
                break;
        cout << i << “,  ”;
}
// կտպի 1, 2, 3, 4
```

```
for(int i = 1; i <= 10; i++)
{
        if( i == 5)
                continue;
        cout << i << “,  ”;
}
// կտպի 1, 2, 3, 4, 6, 7, 8, 9, 10
```

Ֆունկցիաներ

C++ -ում ֆունկցիան հանդիսանում է հիմնական հասկացություն: Ամեն մի ծրագիր անպայման պետք է ունենա main անունով գլխավոր ֆունկցիա, որից էլ սկսվում է

ծրագրի աշխատանքը: Բացի main-ից ծրագրում կարող են լինել նաև այլ ֆունկցիաներ, որոնք կատարում են որոշակի գործողություններ: Ֆունկցիաները օգտագործելուց առաջ պետք է նկարագրված լինեն: Ֆունկցիայի նկարագրման ժամանակ տրվում է գործողությունների հաջորդականությունը, որը պետք է կատարի ֆունկցիան:

Ֆունկցիայի նկարագրությունը ունի հետևյալ տեսքը՝

ֆունկցիայի_տիպ ֆունկցիայի_անուն(ֆորմալ պարամետրերի սպեցիֆիկացիա)
{ֆունկցիայի մարմին}

ֆունկցիայի_տիպ - դա ֆունկցիայի վերադարձրած արժեքի տիպն է: Ֆունկցիան կարող է ոչինչ չվերադարձնել, այդ դեպքում ֆու-այի տիպը գրվում է void

ֆունկցիայի_անուն – նպատակահարմար է ընտրել ունիկալ անուն՝ կատարվելիք գործողությանը համահունչ:

ֆորմալ_պարամետրերի_սպեցիֆիկացիա - նկարագրվում է ֆորմալ պարամետրը: եթե պարամետրեր չկա ապա գրվում է (void) կամ ()

ամեն մի պարամետր նկարագրվում է հետևյալ կերպ՝

տիպ պարամետրերի_անուն

կամ տիպ պարամետրերի_անուն = значение по умалчанию

ֆունկցիայի_մարմին- կազմված է օպերատորների և նկարագրությունների հաջորդականությունից, որոնք վերցված են {}-ի մեջ: Կարևոր օպերատոր է հանդիսանում return(վերադարձի) օպերատորը, որը ունի հետևյալ տեսքը

return արտահայտություն; կամ return;

ֆունկցիան իր աշխատանքը վերջացնում է return օպերատորով(վերադարձնելով արտահայտության արժեքը), իսկ եթե չկա return օպերատորը, ապա ավարտում է՝ կարատելով վերջին օպերատորը:

օր.՝ նակարագրել երկու թվերի max –ը հաշվող ֆունկցիա

```
float max(float a, float b)
{
        float m;
        if(a > b) m = a;
        else m = b;
        return m;
        // կամ ուղղակի return a > b ? a : b ;
}
```

հիմա գրենք ծրագիր որտեղ օգտագործվում է max ֆունկցիան

```
void main()
{
        float f = 4.5; float b(3.4);
        float  e = 7;
        float c = max(f,  b);
        c = max(c, e);

}
```

Ֆունկցիայի նկարագրման ժամանակ a և b կոչվում են ֆորմալ պարամետրեր, իսկ ծրագրում նկարագրված և ֆունկցիային որպես պարամերտր տրված f, b, e, c -երը կոչվում են փաստացի պարամետրեր: Ֆունկցիան դիմելիս ֆորմալ պարամետրերը փոխվում են փաստացի պարամետրերով և ստուգվում են ֆորմալ և փաստացի պարամետրերի համապատասխանությունը, սխալի դեպքում, տրվում է հաղորդագրություն:

օր.՝ նկարագրել ֆունկցիայի, որը վերադարձնում է տրված թվի քառակուսին

```
double sqr(double a)
{
        return a * a;
}
```

ֆունկցիան օգտագործենք ծրագրում

```
void main()
{
        double t = 2, g = 6;
        double  k = sqr(t); // k = t^2
        k = sqr(g); // k = g^2
}
```

Երբ ծրագրում հանդիպում է ֆունկցիայի կանչման օպերատորը, ապա ֆունկցիան սկսվում է բեռնվել: Ֆունկցիայի կանչը կատարվում է երկու ձևով.

1. Եթե ֆունկցիան նկարագրված է որպես inline, ապա կոմպիլատորը ֆունկցիայի մարմինը բերում է և տեղադևում կանչի տեղում
2. Ղեկավարումը տրվում է ֆունկցիային: Եթե ֆունկցաիան ընդունում է պարամետրեր, ապա նրա կանչի ճամանակ պետք է նշված լինեն փաստացի պարամետրերը, նրանք տրվում են ()-ի մեջ և իրարից առանձնացվում են ստորակետով: Ֆունկցիայի աշխատանքը վերջանում է այն ժամանակ, երբ կատարվում է ֆունկցիայի վերջին հրամանը կամ return հրամանը: Դրանից հետո ղեկավարությունը տրվում է ֆունկցիայից հետո հրամանին:

Բոլոր ֆունկցիաները մինչև իրենց կանչը պետք է նկարագրված լինեն: Ֆունկցիայի նկարագրությունը կազմված է ֆունկցիայի վերադարձվող արժեքից, անունից և պարամետրերի ցուցակից: Այդ երեք էլեմենտները կազմում են ֆունկցիայի նախատիպը: Նախատիպը ներկայացնում է ֆունկցիայի ինտերֆեյսը, որի միջոցով ինֆորմացիա ենք ստանում, թե ֆունկցիան ինչ տվյալներ է ստանում և ինչ տվյալներ է վերադարձնում: Եթե ֆունկցիան պարամետրեր չունի գրվում է հետևյալ կերպ՝

int f(); կամ int f(void);

իսկ եթե ֆունկցիան ոչինչ չի վերադարձնում, ապա գրվում է հետևյալ կերպ՝

void f();

Պարամետրերի ցուցակ. այն կազմված է հետևյալ կերպ՝

տիպ պարամետրերի_անուն, տիպ պարամետրերի_անուն1, ….

կամ տիպ պարամետրերի_անուն = значение по умалчанию

Պարամետրերի անունները չեն կարող կրկնվել: Պարամետրերի անունները այնտեղ որտեղ ֆունկցիան հայտարարված է և այնտեղ որտեղ այն նկարագրված է կարող են չհամընկնեն: C++-ում թույլ է տրված ունենալ նույն անունով մեկ կամ մի քանի ֆունկցիաներ, բայց տարբեր պարամետրերի ցուցակով - դա կոչվում է ֆունկցիաների բեռնավորում:

Ֆունկցիայի կանչի ժամանակ կատարվում է ֆորմալ և փաստացի պարամետրերի ստուգում: Ֆորմալ պարամետրը-դա ֆունկցիայի նկարագրման ժամանակ օգտագործվող պարամետրն է: Փաստացին - դա ֆունկցիայի կանչի ժամանակ տրված պարամետրերն են: Եթե այդ երկու պարամետրերի միջև կա անհամապատասխանություն կամ պարամետրերի քանակը չի համապատասխանում, ապա կոմպիլյատորը տալիս է հաղորդագրություն:

օր.՝ ֆունկցիայի նկարագրություն

```
int gumar(int a, int b); // նախատիպ
void main()
{
        int c = 3, d  = 5, e;
        e = gumar(c, d); // ֆունկցիայի կանչ
}

int gumar(int gumar1, gumar2) // բուն ֆունկցիայի նկարագրություն
{ return gumar1 + gumar2 }
```

Ֆունկցիայի արգումենտի տրման ձևերը

Ֆունկցիաները իրենց կատարման ժամանակ ստեկից հիշողություն են օգտագործում: Ստեկի մի մասը տրվում է ֆունկցիային և մինչև ֆունկցիայի աշխատանքի ավարտը հիշողության այդ մասը չի ազատվում: Ֆունկցիայի ամեն մի պարամետրին հիշողություն է հատկացվում: Հատկացված հիշողության չափը(բայթերի քանակը) պայամանավորված է պարամետրեր տիպով: Ֆունկցիայի կանչի ժամանակ հատկացված հիշողությունը բեռնվում է փաստացի պարամետրերի արժեքներով:

Ֆունկցիայի արգումենտի տրման ստանդարտ ձևը արժեքով տրման(передача по значению) ձևն է, որի դեպքում ֆունկցիան չի կարող փոփոխել արգումենտները, որովհետև այս կանչի ժամանակ ստեկում ստեղծվում են այդ արգումենտների կրկնօրինակները, և ֆունկցիան գործ է ունենում այդ կրկնօրինակների հետ: Կրկնօրինակների արժեքների փոփոխությունը ոչ մի ձևով չի ազդում նրա իսկական օբյեկտների արժեքների վրա, և այդ լոկալ կրկնօրինակները ֆունկցիայից դուրս «կորում» են (հասանելի չեն): Երբ ֆունկցիայի արգումենտները մեծ օբյեկտներ են, ապա արգումենտների տրման այս ձևը հարմար չէ՝ կապված հիշողության հետ(ստեղծվում են այդ մեծ օբյեկտների կրկնօրինակներ): Տրման այս ձևը հարմար չէ նաև այն դեպքերում, երբ պոտք է փոփոխել ֆունկցիայի արգումենտները: Ֆունկցիայի արգումենտների տրման հաջորդ երկու ձևերը լրացնում են այս բացը:

```
void f(int a, int b)
{
        int temp = a;
        a = b;
      b = temp;
}
.....
int  a = 4, b = 8;
f(a, b);
cout << a <<"      "<< b; // կտպի  4          8
```

Ֆունկցիայի արգումենտի տրման հղումային(по силке) և ցուցչային(по указателью) դեպքերում ֆունկցիան գործ ունի արդեն բուն այդ պարամետրերի հետ: Ֆունկցիան ստանում է այդ արգումենտների հասցեները և հնարավորություն է ստանում այդ հասցեներում գրված արժեքների հետ կատարել փոփոխություններ: Տրման այս ձևը շատ հարմար է, երբ ֆունկցիային տալիս ենք մեծ օբյեկտներ: Երբ օգտվում ենք հղումային կամ ցուցչային տրման ձևից, ապա ֆունկցիան այդ օբյեկտների կրկնօրինակներ չի ստեղծում(խնայում են հիշողություն): Իսկ եթե ցանկանում ենք, որ այդ արգումենտների հետ փոփոխություն չկատարվի, ապա նրանց նկարագրում ենք որպես const:

օր.՝ `void f(const long& a, const float* b);`

ցուցչային դեպքում

```
void f(int *a, int  *b)
{
        int temp =* a;
         *a = *b;
       *b = temp;
}
.....
int  a = 4, b = 8;
f( &a , &b);  // կանչի ժամանակ ֆունկցիային տեպք է տանք նրանց հասցեները
cout << a <<"      "<< b; // կտպի  8          4
```

հղումային դեպքում

```
void f(int &a, int  &b)
{
        int temp = a;
         a = b;
       b = temp;
}
.....
int  a = 4, b = 8;
f( a , b);
cout << a <<"      "<< b; // կտպի  8          4
```

Հղումի դեպքում, ինչպես հիշում ենք, օբյեկտը պետք է մեկ անգամ պարտադիր ինիցիալիզացիա լինի մեկ այլ օբյեկտի արժեքով, իսկ ցուցիչը կարող է ցույց տա օբյեկտի կամ ոչ (null), դրա համար երբ օգտագործում ենք ցուցչով արգումենտ անպայաման պետք է ստուգել, հավասար է 0-ի: Հղման դեպքում այդ ստուգման կարիքը չկա:

Եթե ֆունկցիայի կատարման ժամանակ արգումենտը պետք է հղվի տարբեր օբյեկտների, ապա ավելի հարմար է ֆունկցիային որպես պարամետր տալ ցուցչային ձևով:

Փոփոխականների տեսանելիության տիրույթ

Այն փոփոխականը, որը սահմանվել է ֆունկցիայի ներսում կոչվում է լոկալ: Տեսանելիության տիրույթը սկսվում է այնտեղից որտեղից այն հայտարարված է մինչև այն բլոկի վերջը, որի մեջ հայտարարված էր: Ֆունկցիայիներից դուրս սահմանված

փոփոխականները կոչվում են գլոբալ: Եթե ֆունկցիայի ներսում սահմանված փոփոխականը ունի նույն անունը ինչ որ գլոբալ փոփոխականը, ապա լոկալ փոփոխականը ծածկում է գլոբալ փոփոխականին: Գլոբալ փոփոխականին դիմելու համար օգտագործում են ::անուն գործողությունը

օր.՝

```
int x = 1; // գլոբալ փոփոխական

void a()
{
        int x = 25; // ամեն անգամ այս ֆունկցիան կանչելիս  x = 25
        cout << "\n in fun. a the lokal varibale x in before changing = " << x;
        x++;
        cout << "\n in fun. a the lokal varibale x in after changing = " << x;
}

void b()
{
        static x = 50; // x-ին սկզբնական արժեքը տրվում է միայն b ֆունկ. առաջին անգամ
        //կանչելիս, իսկ հաջորդ կանչերի դեքում x static-ը պահպանում է իր արժեքը:
        cout << "\n in fun. b the static varibale x in before changing = " << x;
        x++;
        cout << "\n in fun. b the static varibale x in after changing = " << x;
}

void c()
{
        // այս ֆունկցիայում օգտագործվում է գլոբալ x-ը, քանի որ այս ֆունկցիայում չենք
        //նկարագրել մեկ այլ x
        cout << "\n in fun. c the global varibale x in before changing = " << x;
        x++;
        cout << "\n in fun. c the global varibale x in after changing = " << x;
}

void main()
{
        int x = 5; // լոկալ x , քանի որ հայտարարվել է main-ում, այս փոփոխականը
        //անհասանելի է դարձնում գլոբալ x-ը
        cout <<"\nthe lokal variable in main = " << x;
        {
                int x = 7; //երկրորդ լոկալ x-ն է, որն ծածկում է և առաջին լոկալ x-ին և գլոբալ
                //x-ին, այդ x-ը հասանելի է մինչև բլոկի վերջ, այսինքն այս } փակագիծը
        cout <<"\nthe lokal variable in main = " << x;
        }
         cout << "\nthe lokal variable in main = " << x;

        a(); // պարունակում է լոկալ x
```

```
    b(); // պարունակում է static x
    c(); // օգտագործում է գլոբալ x-ը
    a();
    b();
    c();

    cout << þþ\nthe lokal variable in main = “ << x;

}
```

աշխատանքից հետո կտպվի

```
the lokal variable in main = 5
the lokal variable in main = 7 // սա բլոկի մեջի cout-ն է
the lokal variable in main = 5

in fun. a the lokal varibale x in before changing = 25
in fun. a the lokal varibale x in after changing = 26

in fun. b the static varibale x in before changing = 50
in fun. b the static varibale x in after changing = 51

in fun. c the global varibale x in before changing = 1
in fun. c the global varibale x in after changing = 2

in fun. a the lokal varibale x in before changing = 25
in fun. a the lokal varibale x in after changing = 26

in fun. b the static varibale x in before changing = 51 // քանի որ ստատիկ էր, պահպանվել էր
in fun. b the static varibale x in after changing = 52

in fun. c the global varibale x in before changing = 2
in fun. c the global varibale x in after changing = 3

the lokal variable in main = 5
```

Ֆունկցիայի արգումենտի բացակայության(լռելյայն) դեպք

Аргументи по умолчанию

Հիմնականում ֆունկցիային դիմելիս արգումենտներին տալիս ենք որոշակի արժեքներ: Բայց կարելի է արգումենտին արժեք վերագրել լռելյայն(по умолчанию): Դա

արվում է ֆունկցիայի նկարագրության ժամանակ՝ արգումենտին բացահայտ վերագրելով արժեք: Ֆունկցիային դիմելիս այդ արգումենտի բացակայության դեպքում արգումենտի որպես արժեք վերցվում է ֆունկցիայի նկարագրության ժամանակ բացահայտ տրված արժեքը: Եթե արգումենտները շատ են ապա լռելյայն արգումենտը պետք է լինի ամենաաջը:

օր.՝

```
int v(int a = 1, int b = 1, int c = 1 )
{ return a * b * c; }
```

v(1, 1, 1) – ի փոխարեն կարող ենք ուղակի գրել v(), իսկ v(8, 1, 1)-ի փոխարեն կարող ենք գրել v(8), իսկ v(2, 4, 1)-ի փոխարեն՝ v(2, 4)

```
int g(int a, int b , int c = 3)
```

Ֆունկցիաների վերաբեռնում(перегрузка)

C++-ը հնարավորություն է տալիս նկարագրել ֆունկցիաներ, որոնք ունեն նույն անունը, բայց պարամատրերի թիվը, տիպերը տարբեր են և կատարում են տարբեր գործողություններ: Ֆունկցիաների վերաբեռնման միջոցով նույնանուն ֆունկցիան կկատարի տարբեր գործողություններ(և իհարկե կվերադարձնի տարբեր արժեքներ) կախված փաստացի պարամետրերի տիպերից և քանակից:

օր.՝ նկարագրենք max անունով ֆունկցիա երկու և երեք պարամետրերի համար

```
int max(int a, int b)
{
        return a > b ? a : b;
}
int max(int a, int b, int c)
{
        int m = a > b? a : b;
        return m > c ? m : c;
}
```

եթե կանչում ենք max(4,6), ապա բեռնվում է երկու պարամետրով ֆունկցիան, իսկ max(2,7, 5) –ի դեպքում բեռնվում է երեք պարամաետրով ֆունկցիան:

Ֆունկցիաների կաղապար(шаблон)

Ֆունկցիաների վերաբեռնման (перегрузка) –ի ժամանակ պարամետրերի տարբեր տիպերի դեպքում կատարվում եր տարբեր գործողություններ, իսկ եթե տարամետրերի տարբեր տիպերի համար ֆունկցիան կատարում է ճիշտ նույն գործողությունը, ապա

ավելի հարմար է ավելի հարմար է ստեղծել այդ ֆունկցիայի կաղապար(шаблон)՝ տիպը դարձնում ենք կաղապարի պարամետր:

Ֆունկցիայի կաղապար(шаблон) սահմանվում է հետևյալ կերպ՝

template <class կաղապարի_տիպ> ֆունկցիայի նկարագրություն

օր.՝ ենթադրենք մեզ պետք է max ֆունկցիա int և double տիպերի համար, այդ դեպքում մենք պետք է օգտվենք ֆունկցիայի վերաբեռնումից և նույն max անունով ֆունկցիաներ գրենք int և double տիպերի համար:

```
int max(int a, int b)
{
        int m = a > b ? a : b;
        return m;
}
```

```
double max(double a,double b)
{
        double m = a > b ? a : b;
        return m;
}
```

եթե հանկարծ պետք լինի max ֆունկցիան այլ տիպերի համար, ասենք short, long, …., ապա ստիպված բոլոր տիպերի համար պետք է գրենք այդ ֆունկցիան, այսինք վերաբեռնավորենք: Նման դեպքերում հարմար է օգտվել ֆունկցիայի կաղապարից (шаблон) , որը մեր օրինակի համար կունենա հետևյալ տեսքը՝

```
template<class T> T max(T a, T b)
{
        T m  = a > b ? a : b;
        return m;
}
```

այս օրինակում T հանդիսանում է կաղապարի պարամետր, որը կարող է լինել int, double, …: Եթե ֆունկցիային դիմում ենք max(4, 7), այդ դեպքում T պարամետրը դառնում է int, իսկ max(4.5, 3.9)-ի դեպքում T-ն float է;

Ռեկուրսիվ ֆունկցիաներ

Գոյություն ունի երկու տիպի ռեկուրսիա՝ ուղղակի և անուղղակի: Ֆունկցիան անվանում են անուղղակի ռեկուրսիվ եթե նա պարունակում է ուրիշ ֆունկցիայի կանչ, որն էլ պարունակում է այդ ֆունկցիայի կանչը: Այդ դեպում ֆունկցիայի նկարագրության ժամանակ ռեկուրսիան կարող է և չերևա: Ուղղակի ռեկուրսիայի դեպքում ֆունկցիայի ներսում բացահայտ կանչվում է նույն ֆունկցիան:

օր.՝ գրենք ֆակտորիալ հաշվելու ֆունկցիա

```
long fact(int k)
{
        if( k <  0) return 0;
        if(k == 0) retyrn 1;
        return k * fact(k-1);
```

```
}
```

0-ական էլեմենտի համար տալիս 1, իսկ 0-ից մեծ արգումենտների համար կանչում է նույն ֆունկցիան արգումենտ – 1 պարամետրով: k դրական թվի համար կատարվում է

```
k * (k-1) * (k-2) * … * 3 * 2 * 1 * 1
```

fact ֆունկցիայի ռեկուրսիվ կանչերի հաջորդականությունը կանգ է առնում fact(0)-ի դեպքում:

օր.՝ թվի աստիճան հաշվելու ֆունկցիա

```
double pow(double a, int n)
{
        if(n == 0) return 1;
        if(a == 0) return 0;
        if(n > 0) return  a * pow(a, n-1);
        if(n < 0) return pow(a, n + 1) / a;
}
```

pow(3.1, 4) –ի դեպքում կկանչվի pow(3.1, 4) → pow(3.1, 3) → pow(3.1, 2) → pow(3.1, 1) → pow(3.1, 0) –որը կկատարի 3.1 * 3.1 * 3.1 * 3.1 * 1

pow(4.2, -2) –ի համար կկանչվի pow(4.2, -2) → pow(4.2, -1) → pow(4.2, 0) – որը կկատարի 1 / 4.2 / 4.2

Ցուցիչ Ֆունկցիայի վրա

Ցուցիչը, որը ուղղված է ֆունկցիայի վրա ցույց է տալիս հիշողությունում այդ ֆունկցիայի հասցեն: Ֆունկցիայի անունը իրենից ներկայացնում է նրա կոդի(մարմնի) սկզբնական հասցեն: Այդ հասցեն կարելի է տալ ուրիշ ցուցիչի միայն մի պայմանով, որ այդ ցուցիչը պետք է ունենա նույն տիպը ինչ-որ ֆունկցիան(պարամետրերի քանակը և տիպերը պետք է լինեն նույնը): Ցուցիչը, որը ցույց է տալիս ֆունկցիայի, նկարագրվում է հետևյալ կերպ՝

```
տիպ     (*ցուցիչի_անուն)(պարամետրերի ցուցակ);
```

օր.՝

```
double      (*pf)(double d);
```

// նկարագրված է double պարամետրով և double վերադարձնող տիպով ֆունկցիայի ցուցիչ pf անունով:

Ծրագրում pf-ին տալիս ենք որևէ նույն պարամետրերով ֆունկցիայի հասցե, և այդ ֆունկցիային դիմել կարող ենք նաև (*pf)(փաստացի_պարամետրեր) գրելաձևով:

Ֆունկցիայի ցուցչից օգտվել ավելի հարմար է եթե նրան typedef-ի միջոցով տալիս ենք ավելի հարմար անուն:

օր.՝

typedef double (*PF)(double); // double պարամետրով և double վերադարձնող տիպով ֆունկցիայի ցուցիչին տվեցինք PF անունը

PF p; // նշանակում է p-ն, PF տիպի ցուցիչ է, այսինքն double պարամետրով և double վերադարձնող տիպով ֆունկցիայի ցուցիչ է

օր.՝ նկարագրենք Ptr ֆունկցիայի ցուցիչ, և f1, f2 ֆունկցիաներ: Ծրագրի մեջ սկզբից Ptr-ին կվերագրենք f2-ի հասցեն և (*Ptr)()-ի միջոցով կկանչվի f2 ֆունկցիան, որից հետո Ptr-ին վերագրվում է f1-ի հասցեն և այդ դեպքում (*Ptr)()-ի միջոցով կկանչվի f1 ֆունկցիան: Ֆունկցիայի կանչը կարելի կանչել նաև Ptr()-ի միջոցով:

```
#include <iostream.h>

void f1()
{
        cout << "We have Call f1 function\n";
}

void f2()
{
        cout << "We have Call f2 function\n";
}

void main()
{
        void (*Ptr)();
        Ptr = f2; // Ptr-ին տալիս ենք f2 ֆունկցիայի հասցեն
        (*Ptr)(); //կկանչվի f2 ֆունկցիան
        Ptr = f1; // Ptr-ին տալիս ենք f1 ֆունկցիայի հասցեն
        (*Ptr)(); // կկանչվի f1 ֆունկցիան
        Ptr(); // կկանչվի f1 ֆունկցիան
}
```

օր.՝ կնարագրենք մի քանի ֆունկցիա և ցուցիչ այդ ֆունկցիաների տիպի, որից հետո ցուցչին կտանք այն ֆունկցիաների հասցեն, կախված թե ստեղնաշարից ինչ ենք ներմուծում:

```
int add(int a, int b) {return a + b;}
int sub(int a, int b) {return a - b;}
int mul(int a, int b) {return a * b;}
int div(int a, int b) {return a / b;}

typedef int (*PINT)(int a, int b); // ë³ÑÙ³Ý»óÇÝÝ PINT óáõóã³ÛÇÝ ïÇå

void main()
{
        PINT pfun;
```

```
        int a = 10, b = 2;
        char c;
        while (c != 'e')
        {
                cout << "\ninput operation, if you want to exit enter e";
                cin >> c;
                switch(c)
                {
                        case '+' : pfun = add; break; // pfun տալիս ենք add-ի հասեն
                        case '-' : pfun = sub; break; // pfun տալիս ենք sub-ի հասեն
                        case '*' : pfun = mul; break; // pfun տալիս ենք mul-ի հասեն
                        case '/' : pfun = div; break; // pfun տալիս ենք div-ի հասեն
                        default :  cout << "input correct operation"; pfun = add; break;
                }
                cout  <<   a << "  " <<  c  << " " << b ;
                cout << " = " << (*pfun)(a, b); // կամ կարելի նաև pfun(a, b)
        }
}
```

Կարելի է սահմանել ֆունկցիայի վրա ցուցիչների մասիվ, մենյուների հետ աշխատանքը մոդելավորելու համար: Գրենք օրինակ, որտեղ սահմանվում է գործողությունների(ֆունկցիաների) մասիվ ֆունկցիայի վրա ցուցիչի միջոցով, որից հետո օգտագործողի ընտրությունից կախված կատարվում է այդ մասիվի համապատասխան գործողությունը: Օրինակ կատարենք ֆայլերի հետ աշխատող ֆունկցիաների համար:

Նախ սահմանենք ֆունկցիայի ցուցչի տիպ dypedef – ի միջոցով

typedef void (*Menu_Fun)(char*); // սահմանվեց Menu_Fun ֆունկցիայի վրա ցուցիչի տիպ, որը ունի char* պարամետր, և ոչինչ չի վերադարձնում: Այնուհետև կարելի է սահմանել մասիվ այդ տիպի ցուցչիների, և այդ մասիվի էլեմենտներին կարելի է տալ կոնկրետ ֆունկցիաների հասցեներ:

```
void Crate(char* name)
{
        cout << "\nNow is creating the " << name << " file\n";
        // այստեղ կնկարագրվեն ֆայլ ստեղծելու մասը
}

void Delete(char* name)
{
        cout << "\nNow is deleting the " << name << " file\n";
        // այստեղ կնկարագրվեն ֆայլը ջնջելու մասը
}

void Read(char* name)
{
        cout << "\nNow is reading the " << name << " file\n";
```

```
        // այստեղ կնկարագրվեն ֆայլ կարդալու մասը
}

void Change(char* name)
{
        cout << “\nNow is changing the ” << name << “ file\n”;
        // այստեղ կնկարագրվեն ֆայլը փոփոխելու մասը
}

void Close(char* name)
{
        cout << “\nNow is closing the ” << name << “ file\n”;
        // այստեղ կնկարագրվեն ֆայլը փոփոխելու մասը
}

typedef void (*Menu_File)(char* c);

Menu_File Menu[] = { Crate, Delete, Read, Change, Close};
// սահմանեցինք 5-չափանի Menu մասիվը, որի էլեմոնտները ցուցիչներ են  char*
//պարամետրով և ոչինչ ցվերադարձնող ֆունկցիայի վրա void (*Menu_File)(char* c)
//որից հետո այդ մասիվի էլեմենտներին տվել եմ Crate, Delete, Read, Change, Close
//ֆունկցիաների հասցեները
void main()
{
        int number;
        char Filename[30]; // ֆայլի անունը հիշելու համար;
        cout << “\nPlease choose the operation what you want to do”;
        cout << “\n 1 – for crateing file”;
        cout << “\n 2 – for deleting file”;
        cout << “\n 3 – for reading file ”;
        cout << “\n 4 – for changing file”;
        cout << “\n 5 – for closing file”;

        while(1) // անընդհատ աշխատելու համար
        {
                while(1) // այս ցիլը կաշխատի այնքան մինչև չընտրենք 1-5 թվերը
                {
                        cout << “\n\nInput your operatin number:  ”;
                        cin >> number;
                        if(number >= 1 && number <=5 ) break; //դուրս կգա ցիկլից
                }
                if(number != 5)
                {
                        cout << “\n Input the file name”;
                        cin >> Filename;
                }
                Menu[number - 1](Filename); // կկանչի համապատասխան ֆունկցիան
                // որը մենք ընտրել ենք
        }
```

```
}
```

օր.՝ Գրենք ծրագիր, որը գտնում է է f(x) –ի արմատը [a, b] տիրույթում: Օգտվում են տրված հատվածում ֆունկցիայի արմատի գտնման հատվածի կիսման մեթոդից: Մեթոդը մոդելավորող(նակարագրող) ֆունկցիան որպես պարամետր ընդունում է ֆունկցիայի ցուցիչ, որպեսզի հետագայում, տալով կոնկրետ ֆունկցիա գտնենք այն ֆունկցիայի արմատը: Ցուցչի օգտագործելը հարմար է, քանի որ կարող ենք արմատի գտնման ֆունկցիան օգտագործել նաև այլ ֆունկցիաների համար:

```
typedef float (*PF)(float f);

float root(PF F, float a, float b, float eps)
{
        //eps -դա մեթոդի ճշտությունն է
        float x, y, c, Fx, Fy, Fc;
        x = a;
        y = b;
        Fx = F(x);
        Fy = F(y);
        If(Fx * Fy >  0.0)
        {
                cout << “\n Non correct diemtions”;
                exit(1); //
        }
        do
        {
                c = (y - x) / 2; // վերցնում ենք հատվածի կեսը
                Fc = F(c);
                if( Fc * Fx > 0){ x = c; Fx = Fc;}
                else {y = c; Fy = Fc;}
        }while (Fc != 0 && y – x > eps)
         return c
}
float fun1(float f)
{
        return f * f – 2;
}

float fun2(float f)
{
        return f – 5;
}
void main()
{
        float result = root(fun1, 0.0, 3.0, 0.0001); // կգտնի x^2 – 2 ֆունկց. արմատը [0, 3]
        cout << result << endl;
```

```
    result = root(fun2, 0.0, 7.0, 0.0001); // կգտնի x – 5 ֆունկց. արմատը [0, 7]
}
```

Փոփոխական քանակի պարամետրերով Ֆունկցիաներ

C++-ում հանրավոր է նակարագրել ֆունկցիաներ, որոնց պարամետրերի քանակը որոշված չէ, նույնիսկ հնարավոր է, որ պարամետրերի տիպերը հայտնի չլինեն: Փոփոխական քանակով պարամետրերով ֆունկցիաները պետք է ունենան առնվազն մեկ բացահայտ ֆորմալ պարամետր: Պարամետրերի տիպերը և քանակը հայտնի է դառնում միայն ֆունկցիայի կանչի ժամանակ, երբ բացահայտ ձևով տրվում է փաստացի պարամետրերը:

Այդպիսի ֆունկցիայի նախատիպը ունի հետևյալ տեսքը՝

տիպ ֆունկ._անուն(բացահայտ_պար. սպեցիֆիկացիա, ...)

տիպը դա ֆունկցիայի վերադարձվող տիպն է, իսկ բացահայտ_պար. սպեցիֆիկացիա դա պարամետրեր են, որոնք կոմպիլացիայի ժամանակ ֆիքսված են՝ դրանք անվանում են նաև անհրաժեշտ պարամետրեր:

Ամեն մի այդպիսի ֆունկցիայի համար պարամետրերի քանակը և տիպը որոշելու համար պետք է գոյություն ունենա մեխանիզմ: Հիմնական մեխանիզմը դա մի փաստացի պարամատրից մյուսին անցումը կատարվում է ցուցիչի միջոցով:

օր.՝

```
long sum(int k, …)
{
    int* pk = &k;
    long total(0);
    for( ; k; k--) // k-ն դա էլեմենտների թիվն է, -- ով կհասնենք 0-ի
        total += *(++pk); // pk-ն շարժվում է sizeof(int)-ի չափով
    return total;
}

void main()
{
    cout << sum(2, 6, 4);
    cout << sum(4, 2, 4, 2, 3);
}
```

հաջորդ օրինակում պարամետրերի վերջը բնորոշում է 0 -ական էլեմենտը

```
double prod(double arg, …)
{
    double a = 1.0;
    double * pa = &arg;
```

```
        if(*pa == 0.0) return 0.0;
        for( ; *pa; pa++)
                a *= *pa;
        return a;
}
void main()
{
        cout << prod(2e0, 6e0, 4e0, 0e0);
        cout << prod(4.0,  2.0, 4.0,  2.0, 3.0, 0.0);
}
```

Վերը նշված երկու օրինակում պարամետրերի տիպերը ֆիքսված էին, իսկ որպեսզի ֆունկցիան աշխատի տարբեր տիպերի համար օգտագործվում է հետևյալ մեխանիզմը. որևէ «անհրաժեշտ» պարամետրով ներմուծվում է պարամետրերի տիպը

օր.՝

```
long minimum(char c, int k, …)
{
        if(c == 'i')
        {
                int *pi = &k  + 1;
                int min = *pi;
                for( ; k; k--, pi ++)
                        min = min > *pi ? *pi : min;
                return (long)min;
        }
        if(c == 'l')
        {
                long *pl = (long)(&k  + 1);
                long min = *pl;
                for( ; k; k--, pl ++)
                        min = min > *pl ? *pl : min;
                return (long)min;
        }
        // if(c == 'l') կարող ենք գրել նաև մյուս տիպերի համար
}

void main()
{
        cout << minimum ('l', 3, 20L, 234562L, 560943L); // pl –կփողխի sizeof(long) չափով
        cout << minimum ('i',  4, 4,  2, 3, 0, 4); // pi –կփոխվի sizeof(int) չափով
}
```

Փոփոխական քանակով պարամետրերով ֆունկցիաները որպեսզի աշխատեն տարբեր միջավայրերում առաջարկվում է օգտագործել ստանդարտ մակրոհրամաններ,

որոնք նկարագրված են stdarg.h ֆայլում և հնարավորություն են ընձեռնում պարզ և ստանդարտ ձևով դիմել փոփոխական քանակով փաստացի պարամետրերին:

Այդ մակրոհրամանները հետևյալն են

void va_start(va_list param, վերջին_բացահայտ_պարամետր);

type va_arg(va_list param, type);

void va_end(va_list param);

բացի այդ մակրոհրամաներից stdarg.h ֆայլում նկարագրված է նաև va_list տիպը: Ամեն մի փոփոխական քանակով պարամետրերով ֆունկցիա պետք է ունենա va_list տիպի օբյեկտ: Այդ օբյեկտը ունի ցուցիչի հատկություն, և նա կապվում է(ցույց է տալիս) առաջին չնկարագրված պարամետրի հետ, va_start մակրոհրամանի միջոցով, որին որպես պարամետր տրվում է վերջին բացահայտ նկարագրված պարամետրը: va_list տիպի օբյեկտը հանդիսանալով ցուցիչ(որը va_start մակրոհրամանի միջոցով արդեն ուղղված է առաջին չնկարագրված պարամետրի վրա), կարելի է ստանալ մյուս չնկարագրված էլեմենտները օգտագործելով va_arg մակրոհրամանը: Այդ հրամանում type մեզ պետք է տրամադրվի որևէ մեխանիզմով, որը հիմանկանում արվում է պարտադիր ֆորմալ պարամետրերի միջոցով: va_arg մակրոհրամանը առաջին հերթին հնարավորություն է տալիս ստանալ հաջորդ փաստացի պարամետրը, և երկրորդ va_list տիպի ցուցիչը ուղղել հաջորդ փաստացի պարամետրին: va_end մակրոհրամանի միջոցով կորեկտ դուրս են գալիս փոփոխական քանակով պարամետրերով ֆունկցից, կարելի ասել որ այն վերացնում է այդ մակրոհրամաների աշխատանքը:

օր.՝ գրենք տողեր կցելու ֆունկցիա, որը պարամաետրերը քանակը հայտնի չեն

```
#include <iostream.h>
#include <string.h> // տողերի հետ աշխատելու համար
#include <stdarg.h> // մակրոհրամանների հետ աշխատելու համար

char* concat(char *s1, …)
{
        va_list p;
        car* cp = s1;
        int len = strlen(s1); // առաջին պարամետրի երկարությունը
        va_start(p, s1); // փոփոխական ցուցակի սկիզբը
        //այս ցիկլով որոշում ենք փոփոխական քանակով պարամետրերի երկարությունը
        while(cp = va_arg(p, car*) ) // ստանում ենք հաջորդ փաստացի պարամետրը
                len += strlen(cp);
        //len ունենք ընդհանուր երկարությունը, հիմա վերցնենք հիշողություն այդ չափով
        char *string = new char[len+1];
        strcpy(string, s1);
        va_start(p, s1); // նորից գալիս ենք առաջին պարամետրի վրա
```

```
        // հիմա պիտի հերթով կպցնենք փաստացի պարամետրերը
        while( cp = va_arg(p, char*))
                strcat(string, cp) // string կցում է cp
        va_end(p);
        retuen string
}
void  main()
{
        char s = concet(“ Hi ”, “everbody, ”, “How are ”, “You? ”, NULL);
        // s կլինի Hi everybody, How are You?
}
```

ինքնուրույն գրել օրինակ pintf(char* format, …) ֆունկցիան, որին տիպերի մասին ինֆորմացիա ենք տալիս հետևյալ %d և %f նշանակումներով, d-ն նշանակում է տասական թվեր, %f-ն իրական:

Մակրոս

Մակրոսի միջոցով սիմվոլների մի հաջորդականությունը փոխվում է մեկ ուրիշ սիմվոլների հաջորդականությունով: Այդ փոփոխությունը իրականացնելու համար օգտագործվում է հատուկ մակրոհրաման, որը ունի հետևյալ տեսքը՝

```
#define    ինդենտիֆիկատոր    փոխարինվող_մաս
```

կա նաև պարամետրով տեսք

```
#define    անուն(պարամետրերի_ցուցակ)    փոխարինվող_մաս
```

օր.՝

```
#define max(a, b)    (a <  b ? b : a)
```

երբ դիմում ենք max(z, 4), ապա այս տողը կձևափոխվի z < 4 ? 4 : z

```
#define Abs(x)        (x < 0 ? –x : x)
```

Ֆունկցիաները և զանգվածները

Զանգվածները կարող են լինեն ֆունկցիայի պարամետր և ֆունկցիան կարող է վերադարձնի ցուցիչ զանգվածի վրա: երբ ֆունկցիային որպես պարամետր փոխանցում ենք զանգված, ապա ֆունկցիայի մարմնում, այդ զանգվածի էլեմենտների հետ աշխատելիս զանգվածի չափի մասին ինֆորմացիա է անհրաժեշտ: Տողերի հետ աշխատելիս (char []), տողային զանգվածի վերջին էլեմենտը կարելի է ֆիքսել, քանի որ նրանք վերջանում են ’\0’-ական էլեմենտով:

օր.՝ գրենք տողի երկարությունը հաշվող ֆունկցիա

```
int length(char str[])
```

```
{
        int m(0);
        while(str[m++]) ; // m++ կլինի այնքան մինչ str[m]-րդը չլինի '\0'
        return m;
}
```

Այն դեպքերում, երբ ֆունկցիային որպես պարամետր տալիս ենք ոչ տողային տիպի զանգված, ապա պետք է ինչ-որ ձևով տալ նաև նրա չափը: Դա կատարվում է հետևյալ կերպ, կամ այդ ֆունկցիաները աշխատում ենք ֆիքսված չափով զանգվածի հետ, կամ չափը նույնպես տալիս ենք ֆունկցիային՝ որպես պարամետր:

օր.՝

```
void max_vect(int n, int x[], int y[], int z[])
{
        for(int i = 0; i < n; i++)
                z[i] = x[i] > y[i] ? x[i] : y[i];
}
void main()
{
        int a[] = {1, 7, 2, 3, 4, 5, 6, 7};
        int b[] = {7, 6, 5, 4, 3, 2, 2, 5};
        int c[7];
        max_vec(a, b, c);
        for(int i = 0; i < 7; i++)
                cout << c[i] << “, ”;

}
```

Քանիոր զանգվածի անունը իրենից ներկայացնում է ցուցիչ, ապա ֆունկցիային կարելի է որպես պարամետր տալ այդ ցուցիչը: max_vect(int n, int* x, int* y, int* z)

Օրինակով դիտարկենք այն դեպքը, երբ Ֆունկցիան վերադարձնում է զանգված, դա նույն է որ վերադարձնի ցուցիչ այդ զանգվածի վրա:

օր.՝ երկու զանգված կցելու օրինակ, x զանգված n չափանի է, իսկ y-ը m, արդյունքում կստացվի n+m չափանի զանգված, որի ցուցիչն էլ կվերադարձնի ֆունկցիան:

```
int concet(int n, int* x, int m, int* y)
{
        int a = new int[n+m]; // ստեղծեցին, հիշողությունից վերծրեցինք n+m չափի int մասիվ
        for(int i = 0; i< n, i++)
                a[i] = x[i];
        for(i = 0, i < m, i++)
                a[n+i] = y[i];
        return a;
}
 void main()
```

```
{
	int c[] = {1, 4, 3, 6, 8, 3};
	int d[] = {2, 7, 5, 6};
	int * f;
	int n = sizeof(c)/sizeof(c[0]); // c մասիվի էլեմենտների քանակը
	int m = sizeof(d)/sizeof(d[0]); // d մասիվի էլեմենտների քանակը
	f = concet(n, c, m, d);
}
```

Ստանդարտ ֆունկցիաների ցանկ

մաթեմատիկական ֆունկցիաներ – math.h

ֆունկցիա	նախատիպը և կրճատ նկարագրություն
abs	int abs(int i) վերադարձնում է i –ի մոդուլը
fabs	double fabs(double x) վերադարձնում է x –ի մոդուլը
cos	double cos(double x) վերադարձնում է cos(x)–ը, անկյունը տրված է ռադիանով
sin	double sin(double x) վերադարձնում է sin(x)–ը, անկյունը տրված է ռադիանով
tan	double tan(double x) վերադարձնում է tan(x)–ը, անկյունը տրված է ռադիանով
exp	double exp(double x) վերադարձնում է e^x–ը
floor	double floor(double x) վերադարձնում է x չգերազանցող ամենամեծ ամբողջ թիվը
fmod	double fmod(double x, double y) վերադարձնում է x-ը y-ի վրա բաժանելիս մնացորդը
log	double log(double x) վերադարձնում է lnx
log10	double log10(double x) վերադարձնում է $\log_{10}x$
pow	double pow(double x, double y) վերադարձնում է x^y
pow10	double pow10(int p)

	վերադարձնում է 10^p
sqrt	double sqrt(double x) վերադարձնում է քառակուսի արմատ x

տողերի հետ աշխատող ֆունկցիաներ – string.h, stdlib.h

atof	double atof(char* str) վերափոխում է str տողը իրական տիպի
atoi	int atoi(char* str) վերափոխում է str տողը ամբողջ տիպի
atol	long atol(char* str) վերափոխում է str տողը ամբողջ տիպի
itoa	char* itoa(int v, char* str, int base) v ամբողջ թիվը վերափոխում է str տողի, իսկ base-ը բնորոշում է թվերի ներկայացման կարգը(10, 16, 8, 2)
strcat	char* strcat(char* sp, char* si) si տողը վերագրում է sp տողին
strcpy	char* strcpy(char* sp, char* si) si տողը արտատպում է sp
strlen	unsigned strlen(char* str) վերադարձնում է str տողի երկարությունը
strtok	char* strtok(char* str1, const char* str2) str1 տողում փնտրում է str2-ում նշված լեկսեմաները

հատուկ ֆունկցիաներ

ֆունկցիա	**նախատիպը և կրճատ նկարագրություն**	տեղը
delay	void delay(unsigned x) ծրագրի աշխատանքը դադարեցնում է x մվայր.	dos.h
rand	int rand() վերադարձնում է 0- 2^{15}-1 միջակայքի պատահական թիվ	stdlib.h
srand	void srand(unsigned seed) ֆունկցիան ինիցալիզացիա է անում պատահական թվերի գեներատորը	stdlib.h

sound	void sound(unsigned f) արձակում է f Գh հաճախությամբ ձայն	dos.h
nosound	void sound() դադարեցնում է ձայնը, որը սկսել էր sound ֆունկցիան	dos.h

Նախապրոցեսորային միջոցներ կամ Պայմանական կոմպիլացիա

Предпроцесорние средства

Նախապրոցեսորային միջոցները նախատեսված են ծրագրային տեքստը կոմպիլացիայից առաջ մշակելու համար: Օգտագործում են հետևյալ նախապրոցեսորային հրամանները(директиви): Ամեն մի հրաման գրվում է առանձին տողում և սկսվում է # նշանով:

#define	#if	#else	#line
#include	#ifdef	#endif	#error
#undef	#ifndef	#elif	#pragma

#define – նախատեսված է մակրոսներ կամ նախապրոցեսորային ինդետիֆիկատորներ նկարագրելու համար, որոնցից յուրաքանչյուրին տրվում է համապատասխան սիմվոլների հաջորդականություն: Ծրագիր գրելիս կարելի է օգտագործել այդ ինդետիֆիկատորները, որոնք կոմպիլացիայից առաջ փոխարինվում են մեկ ուրիշ սիմվորլների հաջորդականությունով, որը նշել էինք #define –ով:

օր.՝

```
#define begin {
#define end }
```

նախապրոցեսորից հետո

```
void main()                    void main()
begin                          {
    // օպերատորներ                // օպերատորներ
end                            }
```

օր.՝

```
#define str “\nhow are you?”
```

ծրագրում մի քանի տեղ կարող ենք օգտագործել str, որը նախապրոցեսորից հետո կփոխարինվի “\nhow are you?” տողով

```
cout << str; // կտպի how are you?
#define k  40
// k-ի փոխարեն կօգտագործի 40
#define errprint cout << “Error in operation” ;
{
    errprint; // նույնն է ինչ որ գրեինք  cout << “Error in operation” ;
}
```

#undef - այն ինչ սահմանվել է #define-ով, այլևս չի գործում

օր.՝ #define M 16 // M-ը կընդունի 16

```
#undef M       // M-ը կազատվի, և կարող ենք էլի #define նոր արժեք տանք
#define M ՛C՛ // M-ը կընդունի ‘C’
#undef M       // M-ը կազատվի
```

#include – թույլ է տալիս ծրագրային տեքստերում ներառել մեկ այլ ֆայլի պարունակություն, որը նշված է #include ում;

ունի հետևյալ երկու գրելաձևերը

#include <ֆայլի_անուն> և #include ”ֆայլի_անուն”

Եթե ֆայլի անունը < >միջև, ապա նախապրոցեսորը այդ ֆայլը փնտրում է սիստեմային ստանդարտ ֆայլադարանում, իսկ եթե գրված է ” ” –միջև, ապա սկզբից նայում է օգտագործողի ակտիվ ֆայլադարանը և դրանից հետո նոր ստանդարտ ֆայադարանը: Այդ դիրեկտիվի միջոցով հիմանականում ներառում են *.h ֆայլեր, որոնք պարունակում են ֆունկցիաների, կլասների և փոփոխակաների նկարագրություններ:

#if, #ifdef, #ifndef

#else, #endif, #elif –թուլ են տալիս կազմակերպելու ծրագրային տեքստի պայմանական մշակում, այսինքն կոմպիլացիա է արվում ոչ թե ամբողջ տեքստը այլ նրա մի մասը, որը պայմանավորված է այդ դիրեկտիվներով: Ունեն հետևյալ նկարագրությունը՝

```
#if    հաստատուն_արտահայտություն
#ifdef       ինդետիֆիակտոր
#ifdef       ինդետիֆիկատոր
#ifndef      ինդետիֆիկատոր
```

#else

#endif

#elif

առաջին երեք հրամանները կատարում են պայմանի ստուգում, իսկ հաջորդ երկուսով որոշվում են ստուգվող պայամանի ազդման չափը:

օր.՝

```
#if պայման
        տեքստ1
#else
        տեքստ2
#endif
```

եթե պայմանը ճիշտ է, ապա կոմպիլացիա է կատարվում տեքստ1, սխալի դեպքում՝ տեքստ2

օր.՝

```
#if 5
    տեքստ1
#else
   տեքստ2
#endif
```

այս օրինակում տեքստ1-ը, միշտ կոմպիլացիա կարվի, քանի որ #if –ի մոտ գրված է ոչ զրոյական պայման, այսինքն true է:

#ifdef –ի դեպքում ստուգվում է թե արդյոք #ifdef –ի մոտ գրված ինդետիֆիկատորը տվյալ պահին որոշված է #define-ի միջոցով: Որորշված լինելու դեպքում կատարվում է կոմպիլացիայի տեքստ1:

#ifndef-ի դեպքում կատարվում է հակառակ ստուգում, այսինքն ճշմարիտ է համարվում այն դեպքը, երբ իդենֆիկատորը #define-ի միջոցով որոշված չի լինում

օր.՝

```
#define D 1 // որոշեցինք D-ն
// ….. ծրագրային մաս
#ifdef D // ստուգում է D մինչ հիմա որոշված է #define –ի միջոցով
        cout << “Hello”;
#endif
```

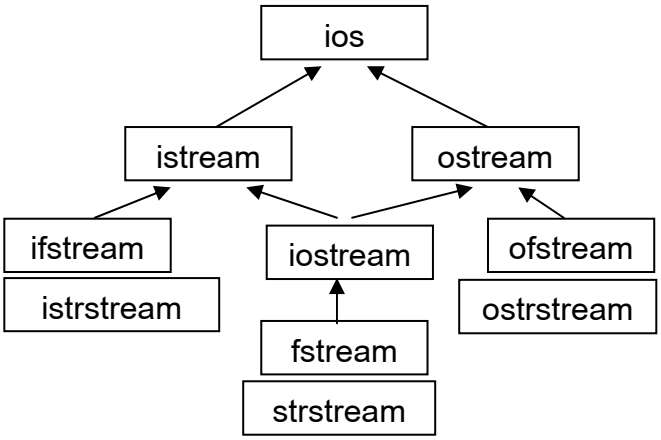
պատասխան. քանի որ D որոշված էր #define –ի միջոցով կտպի Hello

իսկ եթե կախված խնդրի որոշակի պայմանից, եթե չենք ուզում տպենք Hello, ապա մինչև այդ պայամանի ստուգումը կգրենք #undef D

Ֆայլեր

Փոփոխականներն և մասիվները տվյալները պահպանում են ժամանակավոր: Ֆայլները նախատեսված են տվյալների երկարաժամկետ պահպանման համար:

Ֆայլերի հետ աշխատանքի համար անհրաժեշտ է ներառել հետևյալ ֆայլերը <iostream.h> կամ <fstream.h>: fstream.h ֆայլը պարունակում է ifstream ներմուծման, ofstream արտածման, fstream հոսքերի դասերը:



ios - բազային հոսքային դաս
istream - մուտքային հոսքային դաս
ostream - ելքային հոսքային դաս
iostream - մուտքի/ելքի հոսքային դաս
ifstream-մուտքային ֆայլային հոսային դաս
ofstream-ելքային ֆայլային հոսքային դաս
fstream-մ./ել. ֆայլային հոսքային դաս
strstream- մ./ել. տողային հոսքային դաս
istrstream – մուտքի տողային հոսքային դաս
ostrstream - ելքի տողային հոսքային դաս

Ֆայլում ինֆորմացիա գրելու ամենատարածվածը հաջորդական ձևն է, որի դեպքում ինֆորմացիան հաջորդաբար գրանցվում է ֆայլում:

Ֆայլում ինֆորմացիա գրելու համար ստեղծում ենք ofstream դասի օբյեկտ, տալով ֆայլի անունը և բացման ռեժիմը: ինֆորմացիա գրելիս բացման ռեժիմը կարող է լինի կամ ios::out կամ ios::app, որը նախատեսված է տվյալները ֆայլի վերջում կցելու համար: Այդ դասի օբյեկտ կարելի է հայտարարել՝ չտալով ֆայլի անունը և ռեժիմը: այդ դեպքում որևէ ֆայի հետ աշխատելու համար անհրաժեշտ է կանչել այդ դասին պատկանող open ֆունկցիան:

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

void main()
{
        ofstream outFile("Out.txt", ios::out);
        if (!outFile )
        {
                cerr << "Error in file opening\n";
                exit(1);
```

```
    }
    cout << "input name , surname and  numer of telephone\n";

    char name[15];
    char surname[20];
    int number;

    while( cin >> name >> surname >> number )
        outFile << name << " " << surname  << " " << number << endl;
}
```

Ֆայլից ինֆորմացիա կարդալու համար անհրաժեշտ է ստեղծել ifstream դասի օբյեկտ: Հաջորդական ֆայլերի դեպքում որևէ տվյալ փնտրելու համար անհրաժեշտ է ֆայլի սկզբից հերթով կարդալ տվյալները մինչև փնտրվող տվյալը: Կան հատուկ ֆունկցիաններ որոնց միջոցով կարելի է դիրքավորվել: ifstream դասի համար seekg և ostream դասի համր seekp: Դիրքավորումը կատարվում է ֆայլի սկզբի(ios::beg) նկատմամբ, եթե յադ ֆունկցիաններում ոչինչ չենք նշում, և տվյալ դիրքից եթե նշում ենք ios::cur, իսկ ֆայլից վերջից ios::end: tellg և tellp ֆունկցիաները տալիս են ֆայլում գտնվելու դիրքը: Հաջորդական ձևով գրված ֆայլերում որևէ փոփոխություն կատարելու դեպքում հնարավոր է ինֆորմացիայի վնասում(եթե գրվող բառը երկար լինի գրվածից ապա ավելորդ սիմվոլները կգրվեն հարջորդող գրանցված ինֆորմացիայի «վրա»): Նման դեպքերից խուսափելու համար պետք է մինչ այդ գրանցումը արտատպել մեկ այլ ֆայլ, որից հետո գրանցել այդ գրանցումը, որից հետո գրանցել շարունակությունը:

Գոյություն ունի նաև ֆայլում ոչ հաջարդական(կամայական դիմելու) գրանցման եղանակ: Կան այդ գրանցման կազմակերպման մի քանի եղանակ, բայց դրանցից ամենապրիմիտիվը դա կազմակերպել այնպես, որ գրանցվող ինֆորմացիան ունենա միևնույն ֆիքսված չափը: Այս եղանակը հնարավորություն է տալիս ավելի հեշտ գտնել փնտրվող ինֆորմացիան: ostream դասի write ֆունկցիան արտածում ֆիքսված քանակող ինֆորմացիա: իսկ istream դասի read ֆունկցիան էլ ներ է մուծում ֆիքսված քանակով ինֆորմացիա:

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

struct Person
{
    Person(char* n, char* s, int i): Name(n), surName(s), number(i){}

    char Name[10];
```

```
        char surName[15];
        int number;
};

void main()
{
        ofstrem outFile("Out.txt", ios::out);
        if (! outFile)
        {
                cerr << "Error in file opening\n";
                exit(1);
        }

        Person Ara("Aram", "Asatryan", 420308);
        outFile.write( (char*) &Ara, sizeof(Person)  );
}
```

ենթ. ֆայլում կա 150 գրանցում, և մեզ անհրաժեշտ է 20-րդ գրանցումը, ապա դա կարելի է անել

```
        outFile.seekp( (20-1) *  sizeof(Person)  );
```

ֆայլից ինֆորմացիա կարդալու համար կատարում ենք հետևյալը՝

```
        inFile.read( char *) &person, sizeof(Person)  );
```

իսկ ֆայլի վերջը ստուգում ենք

```
        inFile.eof( )
```

ԴԱՍԵՐ(Класс, Class)

Դասը դա տիպ է, որը որոշվում է(սահմանվում է) ծրագարավորողի կողմից արդեն իսկ որոշված տիպերի միջոցով: Դասը իրենից ներկայացնում է տիպերի համախումբ և թույլ է տալիս նկարագրել գործողություններ այդ տիպերի նկատմամբ:

Նկարագրվում է հետևյալ կերպ՝

```
Դասի_բանալի      Դասի_անուն{
        Դասի_կոմպոնենտներ
};
```

Դասի_բանալին - դա հետևյալ բառերից որևիցէ մեկն է՝ class, struct, union:

Դասի_անուն - դա կամայական անուն է, որը բնորոշում է տվյալ դասին

Դասի_կոմպոնենտներ – կարող են լինել տվյալներ, ֆունկցիաներ, դասեր, թվարկում(enum), ընկեր ֆունկցիաներ, ընկեր դասեր:

Դասի ֆունկցիաները անվանում են դասի մեթոդներ կամ կոմպոնենտ ֆունկցիաներ, իսկ տվյալները անվանում են կոմպոնենտներ կամ տվյալ էլեմենտներ:

օր.՝

```
class complex{
public:
        double real;
        double image;

        void define(double r = 0.0, double i = 0.0)
        {
                real = r; image = i;
        }
        void print()
        {
                cout << real << "+ i * " << image << endl;
        }

};
```

այս դասում real-ը և image-ը դասի տվյալներն են, իսկ define և print-ը կոմպոնոնետ ֆունկցիաները: Դասը (օրինակ complex) իրենից ներկայացնում է տիպ և կարելի է նկարագրել այդ տիպի օբյեկտ(փոփոխական), որը կատասրվում է հետևյալ կերպ՝

```
Դասի_անուն      օբյեկտի_անուն;
```

օր.՝

complex x, y ; // x և y complex տիպի են

complex *p; // p-ն complex տիպի ցուցիչ է

complex m[4]; // m-ը 4 չափանի մասիվ է, որի էլեմենտները complex տիպի են

օբյեկտները նկարագրելուց հետո կարելի է դիմել այդ օբյեկտների կոմպոնենտներին (ֆունկցիաներին և տվյալներին), եթե իհարկե նրանք private(գաղտնի) չեն,

complex տիպի x օբյեկտի real կոմպոնենտին դիմելու համար օգտագործում են

x.real գրելաձևը, իսկ define ֆունկցիա-կոմպոնենտին՝ x.define(2,3);

եթե օբյեկտը ցուցիչ է,այդ դեպքում դիմելու համար օգտագործում են -> գործողությունից օր.՝

p->image, կամ p->print()

Դասի տվյալներին դիմելու համար օգտագործում են նաև :: գործեղությունից:

Կոնստրուկտոր և դեստրուկտոր

Դասի նկարագրության ժամանակ դասի տվյալներին սկզբնական արժեք տալու և այդ դասի օբյեկտ ստեղծելու ժամանակ օբյեկտի տվյալներին սկզբնական արժեք տալու համար դասերի նկարագրության ժամանակ պետք է նկարագրել հատուկ կոմպոնենտ ֆունկցիա, որը անվանում են կոնստրուկտոր: Կոնստրուկտորը ունի հետևյալ նկարագրությունը՝

Դասի_անուն(ֆորմալ_պարամետրեր)

{ կոնստրուկտորի_մարմին}

Կոնստրուկտորի անունը պետք է համընկնի դասի անվան հետ և այն ոչ մի արժեք չի վերադարձնում: Կոնստրուկտորի հիմնական և միակ նպատակը օբյեկտների սկզբնական արժեքներ տալն է: Կոնստրուկտորի ֆորմալ պարամետրերի միջոցով հնարավոր է լինում օբյեկտների ստեղծման ժամանակ օբյեկտների տվյալներին փոխանցել իրենց սկզբնական արժեքները: Եթե կոնստրուկտորի մեջ ֆորմալ պարամատրերը նկարագրվել են լռելայն(по умолчанию), ապա կոնստրուկտորին պարամետրեր չտալով այն կվերցնի լռելայն պարամետրերը:

Օբյեկտների ստեղծման ժամանակ, առանց ծրագրավորողի կողմից բացահայտ գրելու, ավտոմատ կանչվում է կոնստրուկտորը: Եթե ստեղծվող օբյեկտի մոտ ( )-ում ոչ մի բան չենք գրում, կամ ( )-եր չենք դնում, ապ կանչվում է կոնստրուկտորը լռելայն պարամետրերով:

Դասը կարող է ունենակ մի քանի կոնստրուկտորներ(օգտագործվում է ֆունկցիաների բեռնավորումը, քանի որ այդ բոլոր կոնստրուկտորները ունեն նույն անունը):

Կոնստրուկտորի հասցեն չենք կարող ստանալ: Կոնստրուկտորի ֆորմալ պարամետրի տիպ չի կարող հանդիսանալ իր դասը(կոնստրուկտորը չի կարող ունենալ իր դասի տիպի ֆորմալ պարամետր), բայց կարելի է օգտագործել այդ դասի հղումը՝ ինչպես արվում է կրկնօրինակող կոնստրուկտորների դեպքում: Կոնստրուկտորը չի կարելի կանչել որպես սովորական կոմպոնենտ ֆունկցիա, այն կանչվում է օբյեկտի ստեղծման ժամանակ, և ունի հետևյալ տեսքը՝

Դասի_անուն օբյեկտի_անուն(փաստացի_պարամետրեր);

Կոնստրուկտորի նկարագրության ժամանակ դասի տվյալների սկզբնական արժեքները տրվում են հետևյալ կերպ՝

Դասի_անուն(պարամետրերի_ցուցակ):սկզբնական_արժեք_ստացող_կոմպոնենտների_ցուցակ

{կոնստրուկտորի_մարմին}

սկզբնական_արժեք_ստացող_կոմպոնենտների_ցուցակ-ը ունի հետևյալ տեսքը՝

կոմպոնենտի_անուն(արտահայտություն);

օր.՝

```
class A{
public:
        int x;
        int y;
A():x(1), y(1){}
};

A a; // a-ն A տիպի է, և a-ի x և y-ը ստեղծման ժամանակ կընդունեն 1
```

Կոմպոնենտների նման ինիցիալիզացիան պարտադիր կատարվում է այն դեպքում երբ՝

- կոմպոնենտը չունի լռելայն կոնստրուկտոր(конструктор по умолчанию)
- կոմպոնենտը const է
- կոնպոնենտը հղում է

օր.՝

```
class X{
        const int i;
        Y y;
        Y& py;
        X(int ii, const Y& cy, const Y& cy1): i(ii), y(cy), py(cy1){}
};
```

մանցած դեպքերում ինիցիալիզացիան կարելի է կատարել նաև կոնստրուկտորի մարմնում՝ վերագրման օպերատորով:

օր.՝

```
class A{
        int a;
        int b;
        A(int aa, int bb)
        { a = aa; b = bb;}
};
```

Օբյեկտների ստեղծման ժամանակ ավտոմատ կանչվում է կոնստրուկտորը, իսկ երբ օբյեկտները դառնում են անհասանելի(դուրս են մնում տեսանելիության տիրույթից, օրինակ երբ օբյեկտը նկարագրվում է բլոկի({ }) ներսում, ապա բլոկից դուրս այն դառնում է անհասանելի) կանչվում է դեստրուկտորը, որը ունի հետևյալ տեսքը՝

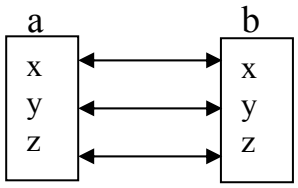
~Դասի_անուն(){դեստրուկտորի_մարմին}

Դեստրուկտորը պարամետրեր չունի և ոչ մի արժեք չի վերադարձնում: Եթե կոնստրուկտորում դինամիկ հիշողություն է զբաղեցվել new օպերատորով, ապա դեստրուկտորում պետք է ազատել այդ հիշողությունը delete օպերատորով: Դեստրուկտորը ապահովում է օբյեկտների «գոյության ավարտը»:

Օբյեկտին նույն տիպի օբյեկտ վերագրելու համար օգտագործում ենք վերագրման օպերատրոը(=): Այդ ժամանակ օբյեկտի կոմպոնենտները ընդունում են վերագրվող օբյեկտի համապատասխան կոմպոնենտների արժեքները, որը որոշ դեպքերում(հիմականում երբ գործ ունենք դինամիկ հատկացված հիշողությունների հետ) հանգեցնում է սխալների և նման դեպքերում պետք է այդ տիպի համար վերաբեռնել = գործողությունը, այսինքն ծրագրավորողը ինքը պետք է գրի թե ինչպես պետք է կատարել վերագրումը:

Ինչպես գիտենք դասը կարող է ունենա մի քանի կոնստրուկտոր, որոնք կատարում են որոշակի գործողություններ: Կրկնօրինակող կոնստրուկտորը նախատեսված է օբյեկտների ստողծման ժամանակ, որպես սկզբնական արժեք նույն դասի օբյեկտ տալու մեխանիզմը կորեկտ կատարելու համար: Երբ տվյալ դասի օբյեկտին որպես սկզբնական արժեք տալիս ենք նույն դասի օբյեկտ ապա կատարվում է տվյալների անդամ առ անդամ ինիցիալիզացիա:

```
class A{        A b;
int x;          A a(b);
int y;
int z;
};
```



Ծրագրավորողը կարող է ինքը կատարել այդ ինիցիալիզացիան՝ գրելով կրկնօրինակող կոնստրուկտոր, որը ունի հետևյալ տեսքը՝

A(const A& x)

ինչպես գիտենք երբ օբյեկտը սահմանում ենք որպես const, ապա այդ օբյեկտի որևէ տվյալ չենք կարող փոփոխել, իսկ եթե նույնիսկ օբյեկտն է հայտարարված const միևնույն է, օբյեկտի այն պարամետրը, որը mutable է կարող է փոփոխվել: Եթե ցանկանում ենք, որ որևէ տվյալ փոփոխվի ապա այն սահմանում ենք որպես mutable:

Դասում ֆունկցիաները և տվյալների համար գոյություն ունի հասանելիության երեք աստիճան՝

public – ֆունկցիաները և տվյալները հասանելի են բոլորին:

private–ֆունկցիաները և տվյալները հասանելի են միայն տվյալ դասի ֆունկցիաներին և այդ դասի ընկեր դասերին

protected-ֆունկցիաները և տվյալները հասանելի են միայն տվյալ դասի կոմպոնենտներին, այդ դասի ժառանգորդ դասերին և ընկեր դասերին:

Եթե դասի նկարագրության ժամանակ բացահայտ չի նշվում այս հասանելիության աստիճանները, ապա class դեպքում լռելյայն վերցվում է private, իսկ struct, union –ի դեպքում՝ public: *class-ի և struct-ի միակ տարբերությունը կայանում է նրանում, որ լռելյայն class-ը վերցնում է private, իսկ struct՝ public:*

Ստատիկ անդամներ

Դասերի ստեղծման ժամանակ որոշակի նկատառումներից ելնելով անհրաժեշտ է լինում ունենալ այնպիսի անդամ, որի արժեքը այդ դասի բոլոր օբյեկտների համար լինի

նույնը: Սովորաբար դասի ամեն մի օբյեկտ ունենում է իր «սեփական» անդամները, որոնք ոչ մի կերպ կապված չեն մյուս օբյեկտների միևնույն անդամի հետ: Ստատիկ անդամները դասի բոլոր օբյեկտների համար «պահպանում» են միևնույն ինֆորմացիան: Ստատիկ անդամները հայտարարվում են static բառով:

Ֆայլի տեսանելիության տիրույթում ստատիկ անդամներին սկզբնական արժեք կարելի է տալ միայն և միայն մեկ անգամ: public ստատիկ անդամների կարելի դիմել այդ դասի կամայական օբյեկտի կամ դասի_անուն:: գործողության միջոցով, իսկ private և protected ստատիկ անդամների դիմելը կատարվում է public ստատիկ ֆունկցիաների միջոցով: Այդ ֆունկցիաները կանչելու համար գրում ենք դասի անունը:: և ֆունկցիայի անունը, կարելի է կանչել նաև ստեղծված օբյեկտի միջոցով:

```
class Student{
public:
        Student(const char* , const char* );
        ~Student();
        const char* getFirstName() const;
        const char* getLastName() const;
        static int getCount();
private:
        char* firstName;
        char* lastName;
        static int Count;
};

int Student::count = 0;
int Student::getCount() {return count;}

Student::Student(const name *first, const char *last )
{
        firstName = new char[strlen(first) +1];
        strcpy(firstName, first);
        lastName = new char[strlen(last) +1];
        strcpy(lastName, first);

        count++;

}

Student::~Student()
{
        delete [ ] firstName;
        delete [ ] last Name;
        count --;
}
```

```
main()
{
        Student *ptr1 = new Student(“Aram”, “Aramyan”);
        Student *ptr2 = new Student(“Anna”, “Asatryan”);

        cout << ptr1->getCount(); // կտպի 2

        delete ptr1;
        cout << Student::getCount(); // կտպի 1

}
```

Const Ֆունկցիաներ և անդամներ

Եթե ցանկանում ենք որ տվյալ դասի որևէ օբյեկտ չպետք է փոփոխվի ապա նրան հայտարարում ենք որպես const: Այդ տիպի օբյեկտների հետ աշխատանքը պետք է ապահովի const ֆունկցիաների միջոցով: Այդ ֆունկցիաները հայտարարվում են const բառով և այն գրվում է ֆունկցիային պարամետրերի ցուցակից հետո: Այդ ֆունկցիաները չեն կարող փոփոխել օբյեկտների անդամները, այլ ուղակի կարող են ցույց տալ նրանց արժեքները:

```
class Date{
        int d, m, y;
public:
        int day() const { return d;}
        int month() const { return m;}
        int year() const { return y;}
};
```

նախատեսված են հիմնականում private կամ protected անդամների մասին ինֆորմացիա ստանալու համար:

Շատ դեպքերում դասի ֆունկցիաները որոնք փոփոխում են դասի անդամները ոչ մի արժեք չեն վերադարձնում: Որոշակի խնդիրների դեպքում հարամար է այդ ֆունկցիաների կանչել շղթայի տեսքով: Որպեսզի ապահովենք այդ հնարավորությունը այդ ֆունկցիաները պետք է վերադարձնեն փոփոխված օբյեկտի հասցեն:

```
class Date{
        // .....
        Date& add_year(int );
        Date& add_month(int );
        Date& add_day(int );
};
Date& Date::add_year(int n)
{
```

```
        if( d == 29 && m == 2 && !leapyear(y+n))
        {
                d = 1; m = 3;
        }
        y += n
        return *this;
}
void f(Date& d, int i)
{
        // ...
        d.add_day(i).add_month(i).add_year(i);
        // ...
}
```

*this տալիս է այն օբյեկտի հասցեն որի համար կանչվել է այս ֆունկցիան:

Mutable անդամներ

Այն անդամները որոնց սկզբում գրված է mutable բառը նշանակում է, որ այդ անդամի արժեքը կարելի փոփոխել նույնիսկ եթե այն հանդիսանում է const օբյեկտի անդամ: mutable նշանակում է ոչ մի դեպքում const չէ:

```
class A{
        mutable int  x;
public:
        void change_x(){ x ++;}
};
A a1;
const A a2;
a1.change_x();
a2.change_x(); // եթե x-ը  mutable չլիներ ապա սխալ կլիներ
```

this ցուցիչը

Ամեն մի օբյեկտ, կարող է որոշի իր հասցեն this ցուցչի միջոցով:

Երբ որևէ օբյեկտի տվյալներ մշակելու համար կանչվում է այդ օբյեկտի դասի ֆունկցիա, ապա այդ ֆունկցիային ավտոմատ կերպով տրվում է այդ օբյեկտի հասցեն, որի համար կանչվել է այդ ֆունկցիան: Ամեն մի օբյեկտ ունի this ցուցիչը, որը ուղղված է դեպի իրեն: Ունի հետևյալ նկարագրությունը՝

Դասի_անուն * const **this** = մշակվող օբյեկտի հասցե

this ցուցիչը բացահայտ ձևով չի կարելի նկարագրել: Նա անփոփոխ է: Հիմնականում նա դասին պատկանող ֆունկցիաներին ցույց է տալիս այն օբյեկտը որի համար այդ ֆունկցիան կանչվել է: this ցուցիչը ֆունկցիաների համար հանդիսանում է լրացուցիչ ինֆորմացիա:

```
class A
```

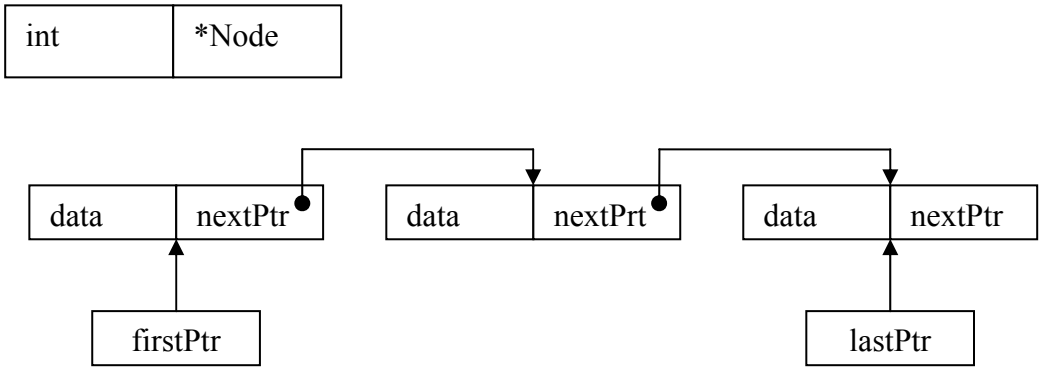
```
{
    int a;
 public:
    void  fn(){ a++; }
};

A x;
x.fn();
```

```
void A::fn(const A* this)
{
   this->a++ ;
}


A::fn(&x);
```

Ընկեր ֆունկցիաներ և կլասներ

Դասի ընկեր ֆունկցիաները չեն հանդիսանում դասի կոմպոնենտ, նրանք նկարագրվում են դասից դուրս, բայց կարող են դիմել դասի private և protected անդամներին: ֆունկցիան չի կարող առանց դասի թույլտվության լինի դասի ընկեր: Այդ թույտվությունը կատարվում է՝ նրան որպես friend նկարագրելով:

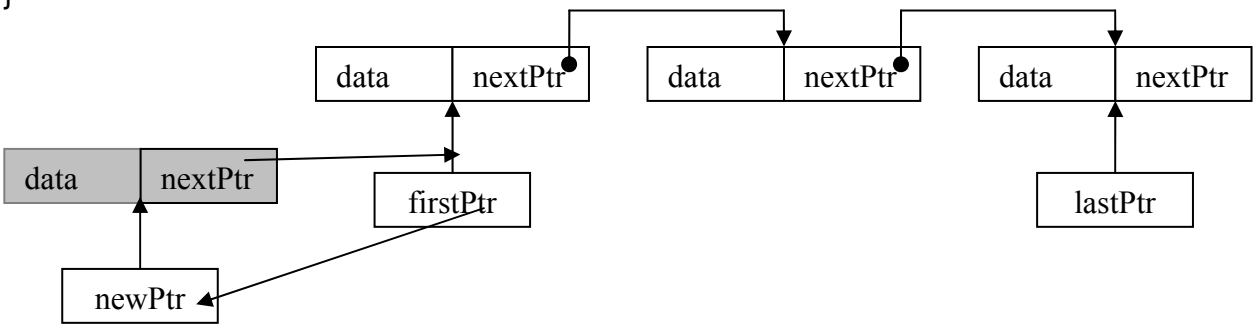
Կապակցված ցուցակի օրինակ

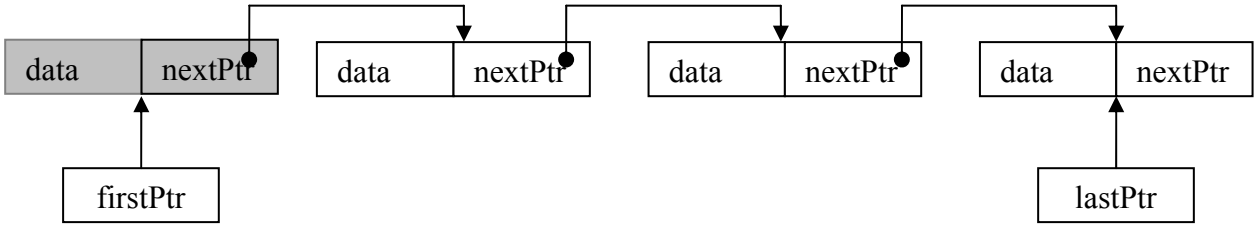



```
class Node{
        friend class List;
public:
        Node(const int& );
        Int getData() const;
Private:
        Int data;
        Node *nextPtr;
};

Node::Node(const int& inf )
{
        data = inf;
        nextPtr = NULL;
}

int Node::getData() const
```

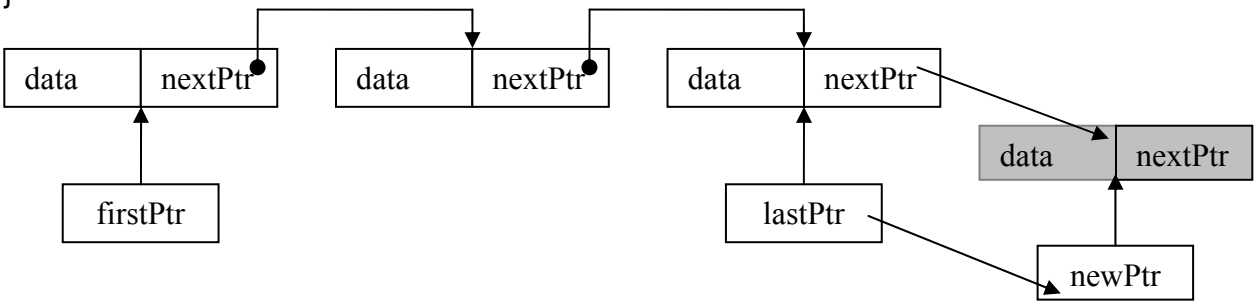
```
{
        return data;
}

class List{

public:
        List();
        ~List();

        void insertAtFront(const int& );
        void insertAtBack(const int& );
        int removeFromFront(int& );
        int removeFromBack(int& );
        bool isEmpty() const;
        void print() const;
private:
        Node *firstPtr;
        Node *lastPtr;

        Node* getNeNode(const int& );
};

List::List(){ firstPtr = lastPtr = NULL; }

List::~List()
{
        if( !isEmpty())
        {
                Node *current = firstPtr;
                Node *tempPtr;

                while( currentPtr != 0)
                {
                        tempPtr = currentPtr;
                        currentPtr = currentPtr->NextPtr;
                        delete tempPtr;
                }
        }
}

void List::insertAtFront(const int& inf)
{
        Node * newPtr = getnewNode(inf);
        if( isEmpty())

                firstPtr = lastPtr = newPtr;
        else
        {
```

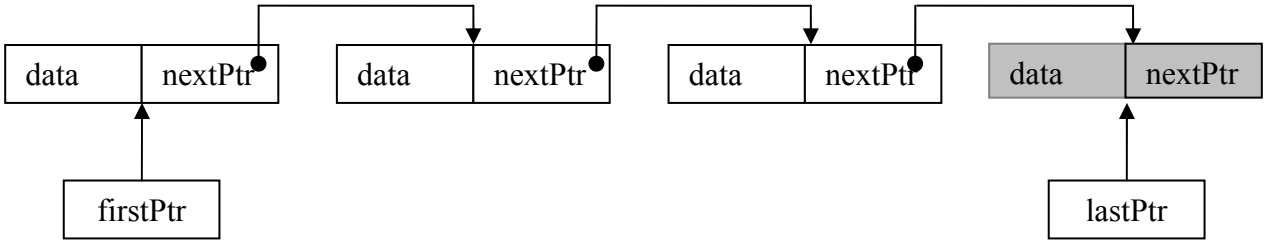
```
            newPtr->nextPtr = firstPtr;
            firstPtr = newPtr;
        }
}
```






```
void List::insertAtBack(const int& info)
{
    Node *newPtr = getNewNode(info);
    if( isEmpty())
        firstPtr = lastPtr = newPtr;
    else
    {
        lastPtr->nextPtr = newPtr;
        lastPtr = newPtr;
    }
}
```





```
bool List::removeFromFront(int& inf)
{
	if( isEmpty())
		reurn false;
	else
	{
		Node *tempPtr = firstPtr;
		if( firstPtr == lastPtr)
			firstPrt = lastPtr = NULL;
		else
			firstPrt = firstPtr->nextPtr;
		inf = tempPtr->data;
		delete tempPtr;
		return true;
	}
}

bool List::removeFromBack(int& inf )
{
	if(isEmpty())
		return false;
	else
	{
		Node *currentPtr = firstPtr;
		while( currentPtr->nextPtr != lastPtr)
			currentPtr = currentPtr->nextPtr;

		lastPtr = currentPtr;
		currentPtr->nextPtr = NULL;
	}

	inf = tempPtr->data;
	delete tempPtr;
	return true;
}

bool List::isEmpty() const { return firstPtr == NULL; }

Node* List::getNewNode(const int& inf)
{
	Node *ptr = new Node(inf);
	return ptr;
}

void List::print() const
```

```
{
	Node* currentPtr = firstPtr;
	while (currentPtr != NULL)
	{
		cout << currentPtr->data << ‘ ’;
		currentPtr = currentPtr->nextPtr;
	}
	cout << endl;
}
```

օրինակ ցուցչի, ստատիկ անդամի և this ցուցչի ընկալման համար:

նկարագրենք տիպ, որի ունի երկու ցուցիչ և մեկ char տիպի փոփոխական: Գոյություն ունի նաև ստատիկ անդամ, որը ցույց է տալիս վերջին անդամին:

```
class Point{
public:
	Point(char c): str(c)
	{
		prev = next = NULL;
	}

	void Add();
	static void print();
private:
	Point *prev;
	Point *next;
	char str;
	static Point *last;
};

int Point::last = NULL;

void Point::Add()
{
	if(last == NULL)
		this->prev = NULL;
	else
	{
		last->next  = this;
		this->prev = last;
	}
	last = this
	this->next = NULL;
}

void Point::print()
{
	Point* temp;
```

```
        temp = last;
        if(temp == NULL)
                cout << “the list is empty\n\a”
        else
           while(temp != NULL)
           {
                cout << temp-> str << “ ”;
                temp = temp->prev;
           }

}

void main()
{
        Point a(‘a’);
        Point b(‘b’);
        Point c(‘c’);
        a.Add(); b.Add(); c.Add();

        Point::print(); // a.print();
}
```

Օպերատորներ

Դասերի միջոցով ստեղծվում են նոր տիպեր և այդ նոր տիպերի համար ստանդարտ գործողության նշանները կիրառելի լինելու համար մտցվում է ստանդարտ գործողությունների բեռնավորում գաղափարը: Ծրագրավորողը պետք է նկարագրի հատուկ ֆունկցիա, որի միջոցով իր կողմից ստեղծված նոր տիպի համար նկարագրի այդ գործողությունը:

օպերատոր ֆունկցիան ունի հետևյալ տեսքը՝

վերադձ._տիպ operator գործողության_նշան(օպերատր_ ֆունկցիայի_պարամետրեր)

C++ -ում թույլատրվում են բեռնավորել հետևյալ գործողությունները

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

հետևյալ գործողությունները չեն թույլատրվում՝

.	*	::	? :	sizeof

Որպեսզի օպերատր ֆունկցիան կապվի դասի հետ ապա օպերատոր ֆունկցիան պետք է լինի կամ տվյալ դասի ֆունկցիա կամ ընկեր ֆունկցիա կամ էլ այդ ֆունկցիայի պարամետրերից գեներ մեկը լինի այդ դասի տիպի:

օրինակ: + գործողության համար օպերատոր ֆունկցիան եթե դասի կոմպնենտ ֆունկցիա է ապա

```
class T{
        //…
        T operator+(const T&  );
};
T a, b;
a+ b; // դա նույն է որ a-ի համար կանչենք a.operator+(b);
```

եթե + գործողությունը նկարագրել են որպես գլոբալ ֆունկցիա

```
T operator+(T a, T b)
{
        // նկարագրություն
}
```

ապա a + b կընդունվի որպես operator+(a, b) ֆունկցիայի կանչ:

օպերատոր ֆունկցիան որպես ընկեր ֆունկցիայի կիրառման վառ օրինակ է >> կամ << գործողությունների բեռնավորումը:

```
class Date{
        // ...
        friend ostream& operator<<(ostream & , const Date& );
        // ...
};
ostream& operator<<(ostream &out, const Date& d )
{
        out << d.day << “/” << d.month << “/” << d.year ;
        return out;
}
```

եթե օպերատոր ֆունկցիայի առաջին պարամետրը ստանդարտ տիպի փոփոխական է, ապա այդ ֆունկցիան չի կարող լինի դասի կոմպոնենտ ֆունկցիա:

ենք. a-ն T տիպի է: a + 2, T տիպի operator+ կոմպոնենտ ֆունկցիայի դեպքում կիասկացվի a.operator+(2); իսկ 2 + a կիասկացվեր 2.operator(a), որը սխալ է քանի որ 2-ը T տիպի փոփոխական չէ, այլ int:

++ և -- գործողությունները ունեն պրեֆիկս և պոստֆիկս (++a, a++): ++ կամ -- գործողությունների բեռնավորելու դեպքում այդ երկու դեպքերի համար պետք է առանձին գրենք:

```
class data{
//...
data operator++(); // ++a;
data operator++(int ); // a++
//...
};
 data data::operator++()
{
// գումարման ալգորիթմ
return *this;
}

data data::operator++(int )
{
data temp = *this;
// գումարման ալգորիթմ
return temp;
}
```

Բինար և ունար գործողությունների համար կարելի է օպերատոր ֆունկկցիան նակարագրել որպես ոչ ստատիկ կոմպոնենտ ֆունկցիա կամ էլ գլոբալ ֆունկցիա երկու պարամետրով:

Կամայական @ բինար գործողությունների համար a@b կընդունվի որպես a.operator@(b) դասի կոմպոնենտ ֆունկցիայի համար և operator@(a,b) գլոբալ ֆունկցիայի դեպքում:

պրեֆիկս բինար @ գործողությամ համար @a կընդունվի a.operator@() դասի կոմպոնենտ ֆունկցիայի կամ operator@(a) գլոբալ ֆունկցիայի, իսկ պոստֆիկս բինար @ գործողությամ համար a@ կընդունվի a.operator@(int) կամ operator@(a, int) գլոբալ ֆունկցիայի դեպքում:

խորհուրդ է տրվում, որ ***operator***=, ***operator***[], ***operator***-> պետք է լինեն դասի ոչ ստատիկ կոմպոնենտ ֆունկցիա

Թվարկումային տիպը նույնպես հանդիսանում է ծրագրավորողի կողմից սահմանած տիպ և նրա համար ել կարող ենք օգտագործել օպերատոր ֆունկցիա

օր. **enum** *Day{sun, mon, tue, wed, thu, fri, sat}*

```
Day& operator++(Day& d)
{
        return d = (sat == d) ? sun : Day( d+ 1);
}
```

Ըստ Ստարուստրուպի դասի մեջ կարելի է նակարագրել այն օպերատորները որոնք ամիջական փոփոխում են առաջին արգումենտի արժեքը(+=, -=): իսկ այն օպերատորները +, -, * կարելի է նակարագրել դասից դուրս

```
class complex{
        double re, im;
public:
        complex& operator+=(complex a)
        {
                re += a.re;
                im += a.im;
                return *this;
        }
        complex& operator+=(double a)
        {
                re += a;
                return *this;
        }

} ;
complex operator+(complex a, complex b)
{
        complex r = a;
        return  r += b;
}

complex operator+(complex a, double b)
{
        complex r = a;
        return  r += b;
}
complex operator+(double a, complex b)
{
        complex r = b;
        return  r += a;
}
```

ընկեր ֆունկցիաների միջոցով կարելի է կազմակերպել մեր ստեղծած տիպերի ներմուծման և արտապատկերման գործողությունները:

```
istream& operator>>(istream& , complex& );
```

*ostream& operator<<(ostream& ,**const** complex&);*

Ժառանգականություն

Ժառանգականությունը թույլ է տալիս ստեղծել նոր դասեր արդեն իսկ գոյություն ունեցող դասերի միջոցով: Այն դասերը որոնցից ստեղծվելու(ծնվելու) են նոր դասեր անվանում են բազային դաս, իսկ նոր առաջացած դասերը անվանում են ժառանգորդ դասեր: Ժառանգորդ դասերը բազային դասերից ժառանգություն են ստանում տվյալներ և ֆունկցաիներ, և կարող են իրենք ել ավելացնել այդ ցանկը և նույնիսկ փոխեն այդ ֆունկցիաների հատկությունը: Ժառանգորդ դասը հիմնականում ավելացնում է իր էլեմենտները և կոմպոնենտ ֆունկցիաները: Ամեն մի ժառանգորդ դասի օբյեկտ հանդիսանում է նաև բազային դասի օբյեկտ, սակայն հակառակը ճիշտ չէ:

Ամեն մի ժառանգորդ դաս իր հերթին կարող է հանդիսանա բազային դաս այլ դասերի համար: այդ դեպքում ստեղծվում է ժառանգման ծառ, որի հիմքում գտնվում է բազային դասը:

Ժառանգման ժամանակ կարևոր դեր է խաղում կոմպոնենտների հասանելիության ստատուսը: Ամեն մի դասի համար դասի կամայական ֆունկցիա կարող է դիմի այդ դասի կոմպոնենտ անդամին և կարող է կանչի այդ դասի ֆունկցիաներին: իսկ դասից դուրս կարելի է միայն դիմել այդ դասի public անդամներին(ֆունկցիաներին):

Ժառանգականության դեպքում դասի private անդամներին հասանելի են միայն տվյալ դասի անդամների համար: proctected կոմպոնենտները հասանելի են դասի պատկանող ֆունկցիաներին և այդ դասից ժառանգված դասի կոմպոնենտներին:

Ժառանգորդ դասի նկարագրման ժամանակ կատարում են հետևյալը

class Դասի_անունը : Բազային_դաս[, Բազային_դաս] { };

եթե : -ից հետո բացահայտ չի գրվում ժառանգման ձևը ապա class դեպքում հասկացվում է որպես private, իսկ struct-ի դեպքում public: Իսկ բացահայտ նշելու դեպքում Բազային դասի անդամները ժառանգորդ դասում ունենում են հետևյալ հասանելիությոան աստիճանները:

Հասանելիության աստիճանը բազային դասում	ժառանգման ձևը	հասանելիությունը ժառանգորդ դասում	
		struct	class

public	-	*public*	*private*
protected	-	*public*	*private*
private	-	*no access*	*no access*
public	*public*	*public*	*public*
protected	*public*	*protected*	*protected*
private	*public*	*no access*	*no access*
public	*protected*	*protected*	*protected*
protected	*protected*	*protected*	*protected*
private	*protected*	*no access*	*no access*
public	*private*	*private*	*private*
protected	*private*	*private*	*private*
private	*private*	*no access*	*no access*

բերել օրինակ private ժառանգման դեպքում, ցույց տալ որ այդ ժառանգման դեպքում բազային դասի ինտերֆեյսը արդեն ժառանգորդի դասում դարձնում են նեքին(թաքցնում ենք), նկատի ունենք, որ արդեն այդ դասի օբյեկտի միջոցով չենք կարոց դիմել այդ անդմեներին կամ ֆունկցաիններին, նրանք միայն տվյալ դասի ներսում են հասանելի:

Քանի որ ժառանգորդ դասը ժառանգում է բազային դասի էլեմենտները, ապա ժառանգորդ դասի օբյեկտ ստեղծելիս անպայման պետք է կանչվի բազային դասի կոնստրուկտորը, համար բազային դասի էլեմենտներին, որոնք կան նաև ժառանգորդ դասում սկզբնական արժեքներ տալու համար: Ժառանգորդ դասի կոնստրուկտորում բացահայտ ձևով կանչելով բազային դասի կոնստրուկտորը կարող ենք տալիս այդ կոնստրուկտորին որոշակի սկզբնական պարամետրեր: եթե բացահայտ չենք կանչում, ապա ժառանգորդ դասի կոնստրուկտորը անբացահայտ ձևով կկանչի բազային դասի լռելայն կոնստրուկտորը: Կոնստրուկտորները և վերագրման գործողության օպերատոր ֆունկցիան չեն ժառանգվում: Սակայն ժառանգորդ դասի կոնստրուկտորները և վերագրման օպերատոր ֆունկցիան կարող են կանչել բազային դասի կոնստրուկտորներին և վերագրման օպերտոր ֆունկցիային: Ժառանգորդ դասի կոնստրուկտորը սկզբից կանչում է բազաին դասի կոնստրուկտորին: իսկ դեստրուկտորներ կանչվում են հակառակ հաջորդականությամբ, այսինքն առաջինը կանչվում է ժառանգորդի դեստրուկտորը հետո նոր բազայինի դեստրուկտորը:
բերենք դասի օրինակ

```
class Point{
public:
        Point(int a = 0, int b= 0)
        {
                x = a; y = b;
                cout << "Constructor of Point";
        }
        ~Point(){ cout << "Destructor of Point"; }

        void setPoint(int a, int b){ x = a; y = b; }
        int getX() const {return x;}
        int getY() const {return y;}
protected:
        int x, y;
};

class Circle : public Point
{
public:
        Circle(int r = 0; int a, int b) : Point(a,b)
        {
                radius = r;
                cout << "Constructor of Circle ";
        }
        ~Circle(){      cout << "Destructor of circle "; }
        void setRadius(int r){ radius = r; }
        int getRadiua(){ return radius; }
protected:
        int radius;
};


void main()
{
        Point *point, p(1, 2);
        Circle *circle, c(2, 3, 6);

        point  = &c;
        circle = (Circle* ) point;

        point = &p;
        circle = (Circle* ) point; // վտանգավոր է

}
```

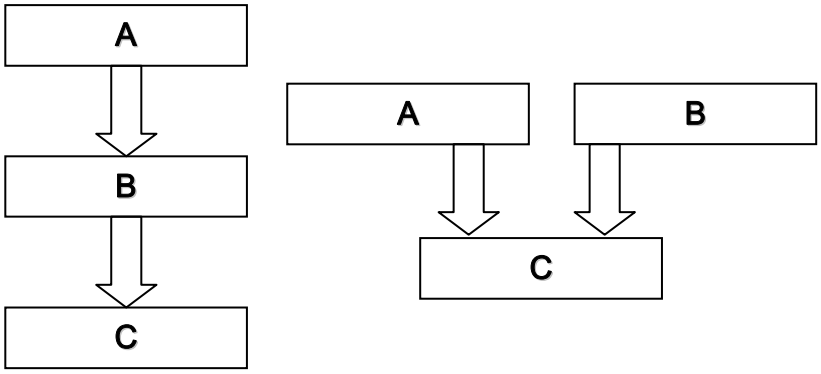
Չնայած որ ժառանգորդ դասի օբյեկտը հանդիսանում է նաև բազային դասի օբյեկտ, սակայն այդ օբյեկտները իրարից տարբեր են: Հիմնականում բազային ժառանգորդ դասը ունի ավելի շատ անդամներ քան բազային դասը: Այդ պատճառով բազային դասի օբյեկը

երբ վերագրում ենք ժառանգորդ դասի օբյեկտին ապա որոշակի անդամները(որոնք չկային բազայինում) կլինեն անորոշ:

Շատ հարմար է ժառանգորդ դասի օբյեկտները դիտել որպես բազային դասի օբյեկտ և նրանց հետ աշխատել բազային դասի ցուցչի միջոցով:

Բազմակի ժառանգում

Դասը կարող է ժառանգել մի քանի դասերից: Ժառանգորդ դասի համար կարող են գոյություն ունենան անուղակի բազային դասեր, որոնք հանդիսանում իրենց բազային դասի համար բազային: Հնարավոր և որ դասը ունենա մի քանի բազային դաս:

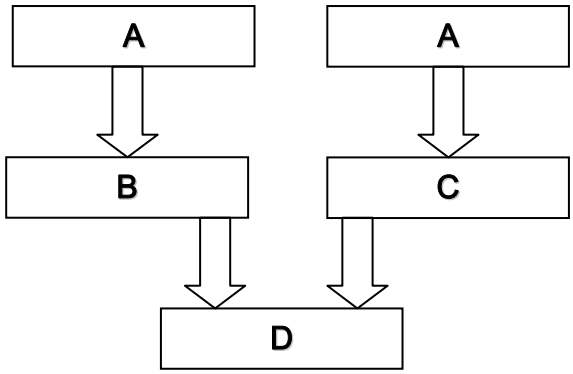



```
class A{ };
class B : public A { };
class C : public B { };
```

```
class C : public A, public B { };
```

Բազմակի ժառանգման ժամանակ ոչ մի դասի չի կարող մեկից ավել անգամ հանդես դա որպես բազային դաս: Սակայն անբացահայտ այդպիսի դեպքեր հնարավոր է:

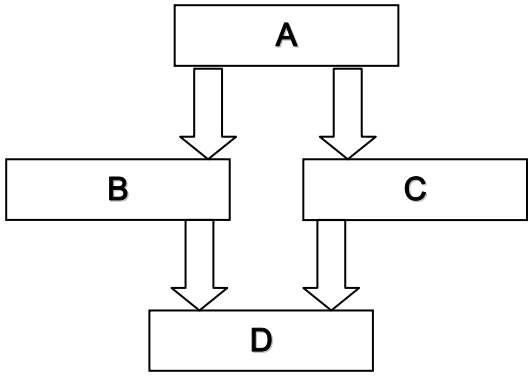
```
class A { f(); };
class B: public A { };
class C: public A { };
class D: public B, public C { };
```



D դասում A բազային դասի f() ֆունկցիային դիմելիս միարժեքությունը խախտվում է, դրա համար պետք է գրել D::C::A::f() կամ D::B::A::f(): Անուղակի կրկնվող բազային դասի միարծեքությունը ապահովելու համար գրվում է virtual բառը:

```
class A { f(); };
class B: virtual public A { };
```

```
class C: virtual public A { };
class D: public B, public C { };
```




```
class Base
{
public:
        Base(int j = 0, char c = '*')
        {
                jj =j; cc = c;
        }
private:
        int jj;
        char cc;
};
```

```
class aBase: public virtual Base
{
public:
        aBase(double d = 0.0) : Base()
        { dd = d;}
private:
        double  dd;
};
```

```
class bBase: public virtual Base
{
public:
        bBase(float f = 0.0) : Base()
        { ff = f; }
private:
    float ff;
};
```

```
class Top: public aBase, public bBase
{
public:
        Top(long t = 0): aBase(), dBase()
        { tt = t; }
private:
        long tt;
};
```

Վիրտուալ ֆունկցիաներ և աբստրակտ դասեր

Վիրտուալ ֆունկցիաների մեխանիզմը արդյունավետ է այն ժամանակ, երբ բազային դասում պետք է նկարագրենք այնպիսի ֆունկցիա, որը տարբեր գործողություններ է կատարում իրեն ժառանգորդ դասերում:

Ենթ. ունենք Shape բազային դասից ստացել ենք Circle Triangle, Rectangle, Square դասերը: Օբյեկտային օրիենտացված ծրագրավորումը թույլ է տալիս, որ այդ դասերից յուրաքանչյուրը ունենա իր draw() ֆունկցիան, որով կնկարի իր ձևը, պարզ է որ տարբեր

պատկերների համար այդ ֆունկցիան պարբեր ալգորիթմ պետք է կիրառի: Կամայական պատկեր նկարելու համար ավելի հարմար է այդ օբյեկտների հետ աշխատել, որպես բազային դասի օբյեկտ: Այդ դեպքում կամայական պատկեր նկարելու համար կանչում ենք Shape բազային դասի draw() ֆունկցիան և կախված թե որ ժառանգորդ դասի օբյեկտի համար է այն կաչվել կատարի համապատասխան գործողություն:

Նման հնարավորություն է ընձեռնվում, եթե բազային դասում draw() ֆունկցիան հայտարարենք որպես virtual:

Եթե draw ֆունկցաին բազային դասում հայտարարել ենք virtual և հետագայում այդ ֆունկցիային դիմում ենք բազային դասի ցուցչի միջոցով, որը ուղղված է ժառանգորդ որևէ դասի օբյեկտին, ապա այդ դեպքում ծրագիրը դինամիկ ձևով ընտրում է ժառանգորդ դասի draw() ֆունկցիան: Նկարագրվածը անվանում են դինամիկ կապակցում, իսկ երբ վիրտուալ ֆունկցիան օգտագործվում է օբյեկտի_անուն . գործողության միջոցով ապա այս դեպքը անվանում ենք ստատիկ և այդ դեպքում draw() ֆունկցիայի ընտրությունը(թե որ ժառանգորդի ֆունկցիան կանչի) կատարվում է կոմպիլացիայի ժամանակ այլ ոչ թե ծրագրի աշխատանքի ժամանակ:

Դասեր նկարագրելիս ենթադրում ենք, որ պետք է գոյություն ունենան այդ դասի օբյեկտներ: Սակայն կան դասեր որոնք չեն ունենում օբյեկտներ, այդպիսի դասին անվանում են աբսկրակտ դասեր և այդ դասերը հիմնականում օգտագործում են որպես բազային դասեր: Հիմնական դերը այդ դասերի դա ստեղծել բազային դաս, որից պետք է ծնվեն նոր դասեր: Դասը անվանում են աբստրակտ եթե նրա ֆունկցաիններից գոնե մեկը մաքուր virtual ֆունկցիա է: Մաքուր virtual ֆունկցիան ունի հետևյալ նկարագրությունը՝

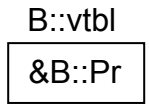
virtual տիպ ֆունկցիայի_անուն(պարամտրեր) = 0;

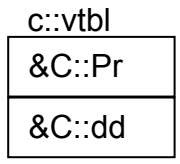
Աբստրակտ դասերը պարտադիր չեն: Երբ ստեղծում են դասերի սերունդ ավելի հարմար է հիմքում ունենալ աբստրակտ դաս:

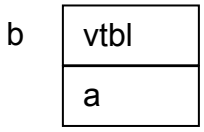
Պոլիմորֆիզմը հնարավորություն են տալիս, որ տարբեր դասերի օբյեկտներ, որոնք ունեն միևնույն բազային դասը(այսինքնժառանգական կապ կա) միևնույն էլեմենտ ֆունկցիային դիմելիս տարբեր գործողություններ կատարեն: Պոլիմորֆիզմը իրականացնում են virtual ֆունկցիաների միձոցով: Բազային դասի ցուցչին տալով ժառանգորդ դասի օբյեկտների հասցեներ ր կանչելով միևնույն էլեմենտ ֆունկցիան դինամիկ ձևով կանչվում է ժառանգորդ դասի համապատասխան ֆունկցիան:

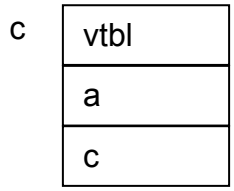
Եթե դասում նկարագրած է virtual ֆունկցիա, ապա տվյալ դասի հետ կապվում է նաև virtual ֆունկցիաների ցուցակ:

```
class B
{
        int a;
        virtual void Pr();
};

class C : public B
{
        int c;
        virtual void Pr();
        virtual void dd();
};

void fn( B* pb)
{
        pb->Pr();
}

main()
{
        B b;
        fn(&b);
        C c;
        fn(&c);
}
```










```
void B::Pr(B* this)
{
   this->a+= 4;
}

void c::Pr(c* this)
{
   this->a+=20;
}

void fn(B* pb)
{
  վերցնում են pb-ի vtbl;
 call(*vtbl +0)(pb)
}
```

Դասերի կաղապար

Կաղապարը հնարավորություն է տալիս ստեղծել այնպիսի դասեր, որոնք կոնկրետ տիպից կախված չեն, և այդ դասի օգտագործման ժամանակ նոր տալիս ենք այդ տիպը: Նկարագրվում նույն ձևով ինչ ֆունկցիաների կաղապարը:

օր.՝

```
template<class T>
class Stack{
public:
        Stack(int = 10;)
        ~Stack() {delete [] stackPtr; }

        int push(const T& );
        int pop(T& );
        int isEmpty() const {return top == -1; }
        int isFull() const { return top == size - 1; }
```

```
private:
        int size;
        int top;
        T* stackPtr;
};

template<class T>
Stack<T>::Stack(int s)
{
        size = s > 0 && s <100 ? s : 10;
        top = -1;
        stackPtr = new T[size];
}

template<class T>
int Stack<T>::push(const T& item)
{
        if ( ! isFull())
        {
                stackPtr[++top] = item;
                return 1;
        }
        return 0;
}

template<class T>
int Stack<T>::pop(T& popVal)
{
        if( ! isEmpty() )
        {
                popVal = stackPtr[top--];
                return 1;
        }
        return 0;
}
```

Նկարագրված օրինակում T-ն հանդիսանում էր կազապարի պարամետր: Կազապարում հնարավորություն կա օգտագործելու նաև ոչ տիպային պարամետր:

օր.՝

```
template<class T, int count>
class Stack{
        //...
private:
        T stack[count];
        //...
};
```

այս դեպքում Stack դասի օբյեկտ ստեղծելի պետք է օգտագործենք հետևյալ գրառումը

```
Stack<int, 20> stack;
```

Բացառիկ դեպքերի մշակում

Բացառիկ դեպքերի մշակումը հնարավորություն է տալիս այս կամ այն բացառիկ դեպքում ծրագրավորողին որոշակի ձևով ղեկավարել դրանց: Բացառիկ դեպք կարող են համարվել 0-ի վրա բաժանումը, հանրահաշվական գործողությունների ժամանակ հիշողության վատնում, մասիվի սահմաններից անցում և այլն:

Բացառիկ դեպքեր մշակելու համար C++ լեզվում մտցված են հետևյալ հրամանները՝ try(ղեկավարում), catch(բռնել), throw(ստեղծել):

try թույլ է տալիս ծրագրային կամայական կտորում կազմակերպել ղեկավարող բլոկ, որն ունի հետևյալ տեսքը՝

try { օպեևատորներ }

այդ օպեևատորների մեջ կարող են լինեն նաև հատուկ օպերատոր, որը ձևավորում է բացառիկ դեպք: Այդ օպերատորը ունի հետևյալ տեսքը՝

throw բացառիկ_դեպքի_արտահայտություն

Երբ կատարվում է այդ օպերատորը, ապա այ դիրամանից հետո գրված արտահայտության միջոցով ձևավորվում է հատուկ օբյեկտ, որը անվանում են Բացաառիկ Դեպք: Այն ձևավորվում է որպես ստատիկ օբյեկտ, որի տիպը որոշվում է Բացառիկ Դեպքի արտահայտությունով: Բացառիկ Դեպքի ձևավորումից հետո throw օպերատորը ղեկավարությունը տալիս է ղեկավարող բլոկից դուրս գտնվող բացառիկ դեպք մշակող ֆունկցիաներին: Ղեկավարող բլոկից դուրս անպայման գտնվում է մեկ կամ մի քանի ֆունկցիաներ, որոնք մշակում են բացառիկ դեպք և ունեն հետևյալ տեսքը՝

catch(բացառիկ_դեպքի_անուն) { օպերատորներ }

օպերատորների միջոցով կատարվում է Բացառիկ դեպքի մշակումը:

գրենք օրինակ, որտեղ գտնում ենք երկու թվերի ամենամեծ ընդհանուր բաժանարարը: այս խնդրի լուծման մի ալգորիթմի պայմանն հետևյալն է, որ այդ թվերը պետք է լինեն 0-ից մեծ:

Ալգորիթմ՝

Ամեն քայլում ստուգվում է թե այդ թվերը հավասար են իրար, հակառակ դեպքում մեծ թվից հանում են փոքր թիվը և տարբերությունը վերագրու մեծ թվին:

```
#include <iostream.h>

int HB(int , int );

void main()
```

```
{
        cout << 48 << "   " <<  63  << HB(48, 36) << endl;
        cout << 0 << "   " << 28  << HB(0, 28) << endl;
        cout << -2 << "   " << 12  << HB(-2, 12) << endl;

}

int HB(int x, int y)
{
        try
        {
                if(x == 0 ||  y == 0 )
                        throw "zero";
                if( x < 0 )
                        throw "1 p<0";
                if( y < 0 )
                        throw "2 p<0";
                while (x != y )
                {
                        if (x > y )
                                x -=  y;
                        else
                                y -= x;
                }
                return x;
        }
        catch(const char *p)
        {
                cerr << p << endl ;
                return 0;
        }
}
```

Շատ դեպքերում ավելի հարմար է նկարագրել դաս, որի միջոցով հնարավոր լինի արտացոլել Բացառկ դեպքերի որոշակի տվյալներ:

```
#include <iostream.h>

int HB(int , int );

class Tv
{
public:
        int x, y;
        char* s;
        Tv(int xa, int ya, char* sa): x(xa), y(ya), s(sa) {}
};

void main()
```

```
{
        cout << 4 << "   " << 16  << HB(4, 16) << endl;
        cout << -3 << "   " << 8  << HB(-3, 8) << endl;
        cout << 2 << "   " << 0  << HB(2, 0) << endl;

}

int HB(int x, int y)
{
        try
        {
                if(x == 0 ||  y == 0 )
                        throw Tv(x, y, "Zero");
                if( x < 0 )
                        throw Tv(x, y, "1 p<0" );
                if( y < 0 )
                        throw Tv(x, y, "2 p<0");
                while (x != y )
                {
                        if (x > y )
                                x -=  y;
                        else
                                y -= x;
                }
                return x;
        }
        catch(Tv a)
        {
                cerr << a.x << a.y << a.s << endl ;
                return 0;
        }
}
```

Եվ այսպես բացառիկ դեպքերի մշակման միջոցով ծրագրավորողին հնարավորություն է տրվում դինամիկ ձևով կատարել առաջացող բացառիկ դեպքերի մշակում, որոնք չեն կարող ղեկավարվել այլ ֆունկցիաների միջոցով, որտեղ ստեղծվել են այդ դեպքերը և այդ ֆունկցիան ձևավորելով բացառիկ դեպք ղեկավարումը տալիս է մեկ այլ ֆունկցիայի, որը կոչված է մշակելու այդ դեպքը: Ընդհանուր դեպքում կարող են ձևավորվել տարբեր տիպի բացառիկ դեպքեր և համապատասխանաբար նրանց մշակող ֆունկցիաներն են կարող են լինեն մի քանի հատ, որոնք դասավորվում են հաջորդաբար:

```
catch(...){ // ...  }
```

այս ֆունկցիան անկախ բացառիկ դեպքի տեսակից պատասխանում է բոլոր throw օպերատորով ծնված բացառիկ դեպքերին և այդ ֆունկցիան գրվում է ամենավերջում:

Բացառիկ դեպք մշակող ֆունկցիան throw օպերատորի միջոցով կարող է նորից առաջացնել բացառիկ դեպք: