

5. СЪСТАВНИ ПРОГРАМИ

5.1. ПРОСТИ И СЪСТАВНИ ПРОГРАМИ

До тук разглеждахме програми, които се състояха от една функция – функцията `main`. Такива програми ще наричаме прости. Повечето програми, които се налага да се създават за нуждите на практиката, обаче, са съставни, т.е. състоящи се от множество функции. Нещо повече, добрият стил на програмиране изисква програмите да се изграждат като множество от неголеми функции, като всяка функция решава определена подзадача от задачата, решавана от цялата програма. Т.е. отделните функции са подпрограми в цялата програма, а сред тях функцията `main` е главна. Тя определя входната точка в програмата, т.е. изпълнението на програмата започва от нея. Тя може да се обръща към другите функции (да ги извиква, да активира изпълнението им), но те не могат да се обръщат към нея.

Този стил на програмиране е свързан със следните предимства:

- програмата става по-прегледна и по-ясна;
- по-лесно става тестването, настройка и модификация на програмата;
- програмата става по-кратка, тъй като многократно повтарящите се нейни фрагменти се обособяват като функции, които се дефинират еднократно, и след това се извикват многократно;
- постига се икономия на памет, тъй като кодът на дадена функция се съхранява само на едно място в паметта, независимо от броя на нейните извиквания (изпълнения);
- разработването на всяка функция от една голяма програма може да се възложи на отделен програмист и по този начин да се съкрати времето за разработването ѝ.

Относно подреждането на функциите в една съставна програма няма ограничения. Трябва обаче да се знае, че функцията `F1` може да се обърне към функцията `F2`, ако функцията `F2` или нейния прототип са разположени пред функцията `F1`. Прототип на функцията се нарича нейното заглавие. В прототипа могат да се пропуснат имената на фиктивните аргументи. Прототипът трябва да завършва със знака `';`.

5.2. ФУНКЦИЯ

5.2.1. Дефиниция на функция

Дефиницията на функцията се състои от две части: заглавие (прототип) и тяло. Общият вид на дефиницията на функция е следният:

```
[Модификатор] [Тип] Име_на_функ ([Списък на аргументите])  
{  
    Тяло на функцията (множество оператори)  
}
```

Модификатор. Такъв може и да няма. Но, ако все пак го има, такъв може да бъде някоя от ключовите думи `inline`, `overload`, `virtual`, `static` и др. Тяхното действие е изяснено по-нататък в тази глава.

Тип на функцията. Обикновено функция се създава за да намери някаква стойност, наречена стойност на функцията. Типът на функцията трябва да съответства на типа на стойността на функцията. Тук ще създаваме функции, чийто тип е някой от аритметичните типове (`int`, `char`, `float` и `double`). По-късно, в следващите глави, ще разгледаме и други възможности. Ако типът на функцията не е указан, трябва да се подразбира `int`.

В C++ могат да бъдат дефинирани функции без стойност, т.е. функции, които не придобиват и не връщат стойност. Тази възможност е предвидена, тъй като има алгоритми, които не са свързани с получаването на една стойност. Такива са например размяната на стойностите на две променливи, сортирането (подреждането) на елементите на едномерен масив в нарастващ или намаляващ ред, въвеждането или извеждането на някакви данни. Функции, които не връщат стойност са от тип `void`. В някои езици за програмиране такива функции се наричат процедури.

Име на функцията. Избира се от програмиста в съответствие с правилата за избиране на имена в C++.

Списък на аргументите. Аргументите на функцията се наричат още фиктивни аргументи, формални аргументи, фиктивни параметри, или формални параметри. Една част от тях служат за да предадат на функцията данните, които ѝ са нужни за да изпълни задачата си, т.е. те са входните аргументи на функцията. Друга част служат да направят стойности, пресметнати във функцията, достъпни за функцията, която е извикала извиканата функция. Такива аргументи са изходни. Има и входно-изходни аргументи. Те се използват както за предаване на данни на функцията, така и за връщане на получени в нея резултати.

Фиктивните аргументи са два вида: обекти и псевдоними на обекти. Тук под обекти ще разбираме константи и променливи от числов или символен тип, както и константи и променливи от тип указател, например

```
void F ( int n, const float a, double b, char c, int* p, int &m, float &x, const double &y, char &z, int* &q),
```

В списъка фиктивните аргументи са изброени заедно с техните типове и са разделени със запетаи. Аргументите, които включват знака `&` са псевдоними. Целият списък е ограден в обикновени скоби. Скобите присъстват и когато списъкът е празен. На аргументите константи или псевдоними на константи не могат да се присвоява стойност в тялото на функцията.

Тяло на функцията. Състои от множество оператори, поместени между две фигурни скоби. Тялото може да съдържа дефиниции на

променливи или константи и оператори, описващи алгоритъма, изпълняван от функцията. Променливите, дефинирани във функцията, се наричат **вътрешни променливи**. Те са достъпни само за операторите на функцията и се използват за съхранение на междинни резултати във функцията.

Фиктивните аргументи и вътрешните променливи и константи се наричат **автоматични променливи и константи**.

5.2.2. Обръщение към функция

Употребяват се и понятията извикване на функцията и активиране на функцията. Както вече пояснихме, функцията F1 може да се обърне към функцията F2, ако функцията F2 или нейния прототип са разположени пред функцията F1. Обръщението включва името на функцията и действителните аргументи. Последните се наричат още фактически аргументи, действителни параметри или фактически параметри. В обръщението, на всеки фиктивен аргумент трябва да съответства действителен аргумент. Връзката между фиктивен и действителен аргумент е различна за аргументите обекти и аргументите псевдоними.

Фиктивни аргументи обекти. При обръщение към функция на такъв фиктивен аргумент може да се съпостави константа, променлива или израз от конвертируем тип, т.е. от тип, който може да се преобразува до типа на фиктивния аргумент. Програмистът обаче трябва да се съобразява с резултата от преобразуването. Например, ако фиктивният аргумент е от тип `int` и му се съпостави като действителен аргумент нецяло число, последното ще се преобразува в цяло, като се пренебрегне дробната му част. На аргумент обект може да се присвои стойност във функцията. С това обаче няма да се промени стойността на действителния аргумент, съпоставен на фиктивния в обръщението към функцията. Следователно фиктивните аргументи обекти могат да изпълняват ролята само на входни аргументи.

Фиктивни аргументи псевдоними. Те могат да се използват като входни, изходни и входно-изходни. Когато се използват като входни аргументи, те не се различават от аргументите обекти – могат да им се съпоставят константи, променливи и изрази от конвертируем тип. Когато се използват като изходни или входно-изходни аргументи, трябва да им се съпоставят променливи от същия тип. Допускат се някои изключения, но е по-добре да не се използват, защото могат да станат причина за трудно откриваеми грешки.

5.2.3. Изпълнение на съставна програма

Изпълнението на съставна програма започва от главната програма. Когато се достигне до обръщение към функция, изпълнява се следното:

- в програмния стек се създават фиктивните аргументи обекти и се инициализират със съпоставените им стойности;
- в програмния стек се създават и инициализират фиктивните аргументи псевдоними, на които в обръщението е съпоставена константа, израз или обект от тип, различен от техния тип;
- всеки фиктивен аргумент-псевдоним се свързва със съответстващия му действителен аргумент променлива, за да може функцията да работи с действителния аргумент чрез неговия псевдоним;
- изпълняват се операторите на функцията, ако някой оператор дефинира обект, последният се създава в също в програмния стек;
- излиза се от функцията;
- унищожават се блокът от данни в програмния стек, принадлежащи на функцията;
- обръщението към функцията се замества със стойността на функцията;
- продължава изпълнението на главната функция.

От това описание не трябва да се остава с впечатлението, че само главната функция може да се обръща към функции от програмата, защото това не е вярно. Всяка функция може да се обърне към всяка функция, освен към главната.

5.2.4. Излизане от функция

Когато функцията изпълни задачата, заради която е направено обръщение към нея, следва излизане от нея. От функция се излиза в два случая:

- когато се достигне край на функцията, т.е. когато се стигне до фигурната скоба, затваряща тялото на функцията;
- когато се достигне оператор `return`.

Синтаксисът на оператора `return` е следният

```
return [a];
```

където `a` е константа, променлива или израз от конвертируем тип, т.е. от тип, позволяващ преобразуване до типа на функцията. Изпълнението на оператора `return a` се изразява в изчисляване на израза `a`, преобразуване на изчислената стойност до типа на функцията, превръщане на преобразуваната стойност в стойност на функцията и излизане от функцията.

Функциите от тип `void` не завършват със стойност и поради това операторът `return` при тях се използва без израза `a`, т.е. във вида

```
return;
```

Във всяка функция може да има повече от един оператор `return`.

Програма 5.1. Функция за намиране на по-голямото от две цели числа.

```
int max2(int x, int y)
{
```

```

int z;    //z - вътрешна променлива
if ( x > y ) z = x;
else z=y;
return z;    //Връщане стойността на z, като
             // стойност на функцията
}

```

Тук е дефинирана функция, която има два входни аргумента (две входни данни) x и y от целочислен тип и една вътрешна променлива z за съхранение на междинен резултат. Функцията намира по-голямата от стойностите на x и y и чрез оператора `return` я връща като стойност на функцията `Max2`.

Активирането (извикването) на функция става чрез нейното име, като фиктивните аргументи се заместят с фактически, например:

```

max2 (2, 3) ;
m=3+max2 (2, 3) ;

```

В този фрагмент функцията `max2` е извикана два пъти с фактически параметри съответно 2 и 3. При първото извикване върнатата стойност на функцията не се използва, т.е. извикването и е безпредметно. При второто извикване, функцията `max2` участва в аритметичен израз. Нейната върната стойност ще бъде добавена към числото 3 и резултатът ще бъде присвоен на променливата m . Следователно m ще получи стойност 6.

5.2.5. Декларация на функция

Декларацията на функция представлява нейното заглавие, но допълнено с точка и запетая. Например, декларацията на функцията `Max2` изглежда така:

```
int max2(int x, int y);
```

В декларацията на функцията е задължително задаването само на типовете на фиктивните аргументи, имената им могат да се пропускат. Следователно функцията `Max2` може да се декларира и по следния начин:

```
int max2(int , int );
```

Функцията трябва да се декларира, само когато е разположена след функция, която се обръща към нея, и декларацията трябва да е разположена преди първата функция, която я извиква, както е показано в следната програма:

5.2.6. Няколко примерни програми с функции

При създаване на съставна програма на програмистът му се налага да вземе следните решения за всяка функция:

- от какъв тип да е функцията;

- колко аргументи трябва да има функцията;
- от какъв тип да е всеки фиктивен аргумент;
- от какъв вид да е всеки фиктивен аргумент – обект или псевдоним на обект.

Програма 5.2. Програма с функция, за намиране по-голямата от две зададени стойности.

Естествено е функцията да получава двете стойности чрез своите аргументи, а намерената по-голяма стойност да е стойност на функцията. Ако допуснем, че зададените стойности са целочислени, то и по-голямата от тях, т.е. стойността на функцията, ще е целочислена. Така достигаме до решението функцията да е от целочислен тип и да има два целочислени аргумента.

Задачата на аргументите е само да предадат на функцията двете стойности, т.е. те са входни аргументи и следователно могат да бъдат аргументи обекти.

По-долу са дадени два варианта на програмата.

- I –ви вариант – с използване на вътрешна променлива:

```
#include <stdio.h>
//Дефиниция на функцията max2
int max2(int x, int y)
{
    int z;
    if (x>y) z=x;
    else z=y;
    return z;
}
//Дефиниция на функцията main
void main()
{
    int a,b;
    printf("a=");scanf("%d",&a);
    printf("b=");scanf("%d",&b);
    printf("По-голямото е: %d\n",max2(a,b));
}
```

- II –ри вариант – без използване на вътрешна променлива:

```
#include <stdio.h>
//Дефиниция на функцията max2
int max2(int x,int y)
{
    if (x>y) return x;
    return y;
}
//Дефиниция на функцията main
void main()
{
```

```

int a,b;
printf("a=");scanf("%d",&a);
printf("b=");scanf("%d",&b);
printf("По-голямото е: %d\n",max2(a,b));
}

```

Програма 5.3. Програма с функция за размяна на стойностите на 2 променливи.

Функцията трябва да е от тип void, защото не е свързана с получаването на някаква стойност. Тя трябва да има два аргумента и те трябва да са псевдоними, защото са входно-изходни аргументи. При обръщение към функцията аргументите псевдоними ще бъдат псевдоними на променливите, на които трябва да размени стойностите.

```

#include <iostream.h>
//Дефиниция на функция change
void change( int &x, int &y )
{
    int buf;
    buf = x;
    x=y;
    y=buf;
}
//Дефиниция на функция main
void main()
{
    int p=2,q=3;
    printf("Преди размяната: p=%d q=%d\n",p,q);
    change(p,q);
    printf("След размяната: p=%d q=%d\n",p,q);
}

```

5.2.7. Указатели като фиктивни аргументи на функции

Функциите могат да имат за аргументи указатели и псевдоними на указатели. При обръщение към такава функция на указателите се съпоставят адреси на променливи или константи, а на псевдонимите на указатели се съпоставят променливи указатели. Тук ще разгледаме две примерни програми.

Програма 5.4. Функция с фиктивни аргументи променлива и указател

```

#include <stdio.h>
//Дефиниция на функцията func1
void func( int x, int *y)
{
    x++; *y=x;
}
//Дефиниция на функцията main

```

```

void main()
{
    int a=1, b;
    func(a, &b);
    printf("След функцията -> a=%d, b=%d\n", a, b);
}

```

Функцията в горната програма е с два аргумента-обекти. Вторият аргумент обаче е указател. В обръщението към функцията му е съпоставен адресът на b. При изпълнението на функцията в програмния стек се създава аргумента-променлива x и се инициализира със стойността на a. Създава се и аргумента-указател y и се инициализира с адреса на b. При изпълнението на операторите от тялото на функцията се работи със съдържанието на указателя y, а такава е стойността на b, т.е. когато във функцията се променя съдържанието на y, всъщност се променя стойността на b. След излизане от функцията, b е с променената във функцията стойност. От тази примерна програма се вижда, че от функциите могат да се връщат стойности не само чрез аргументи псевдоними, а и чрез аргументи указатели. В програмния език C, който не предлага псевдоними, за връщане на стойности, получени във функциите, се използват указателите.

Програма 5.5. Функция за размяна на стойностите на 2 променливи чрез аргументи-указатели

```

#include <stdio.h>
void change (int *x, int *y)
//x и y са указатели към int
{
    int buf;
    buf = *x; *x = *y; *y = buf;
}

void main() {
    int p=2, q=3 ;
    printf("Преди размяната:p=%d q=%d\n", p, q);
    change(&p, &q); //адресите на p и q
    printf("След размяната:p=%d q=%d\n", p, q);
}

```

Програма 5.6. Програма с функция за намиране на квадратен корен

В литературата е известен алгоритъма на Нютон за решаване на тази задача. Той е итеративен и се изразява в следното

- приемаме за начално приближение на търсения корен $y_n = x/2$;
- приемаме новото приближение за старо ($y_s = y_n$) и намираме следващо ново приближение по формулата $y_n = (y_s + x/y_s)/2$. Това

продължава докато $|y_n - y_s| > \varepsilon$, където ε е точността, с която се търси корена.

```
#include <stdio.h>
#include <math.h>
float koren2(float x)
{
    float ys, // - старо приближение на търсения корен
          yn; // - ново приближение на търсения корен
    yn=x/2;   // - начално приближение на търсения корен
    do {
        ys=yn;
        yn=0.5*(ys+x/ys);
    } while (fabs(yn-ys)>0.0000001);
    return yn;
}
void main()
{
    float X;
    printf("Въведете положително число: "); scanf("%f",&X);
    printf("Коренът на %f е %f\n",X,koren2(X));
}
```

Програма 5.7. Програма с функция за намиране на синуса на ъгъл от 0 до $\pi/2$ радиани по формулата

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

```
#include <stdio.h>
#include <math.h>
float sinus(float x)
{
    int i=1;
    float s=0, a=x, x2=x*x;
    while (fabs(a)>0.0000001) {
        s+=a; i+=2;
        a=a*(-x2/(i*(i-1)));
    }
    return s;
}
void main()
{
    float x;
    printf("Въведете ъгъл от 0 до pi/2 радиани: ");
    scanf("%f",&x);
}
```

```

    printf("Синусът на ъгъл %f радиани е %f\n",sinus(x));
}

```

Програма 5.8. Програма за обръщане реда на цифрите в число

```

#include <stdio.h>
#include <math.h>
long inv(long n)
{
    long N=0; short c;
    while (n>0){
        c=n%10;
        N=10*N+c;
        n=n/10;
    }
    return N;
}
void main()
{
    long m;
    printf("m="); scanf("%ld",&m);
    printf("%ld\n",inv(m));
}

```

Програма 5.9. Програма с функция за сравняване на дати и функция за въвеждане на дата.

```

#include <stdio.h>

//Функция за сравняване на дати
int datcmp(int d1, int m1, int g1,
           int d2, int m2, int g2)
{
    if ( g1 != g2 ) return g1-g2;
    if ( m1 != m2 ) return m1-m2;
    return d1-d2;
}
void readdat(int *d, int *m, int *g)
{
    printf("    - ден: "); scanf("%d",d);
    printf("    - месец: "); scanf("%d",m);
    printf("    - година: "); scanf("%d",g);
}
void main()
{
    int D1, M1, G1, D2, M2, G2;
    printf("Въведете първата дата:\n");
}

```

```

readdat (&D1, &M1, &G1);
printf("Въведете втората дата:\n");
readdat (&D2, &M2, &G2);
int n=datcmp(D1, M1, G1, D2, M2, G2);
if ( n<0 )
    printf("Първата дата е по-ранна\n");
else if ( n==0 )
    printf("Двете дати съвпадат.\n");
    else
    printf("Първата дата е по-късна\n");
}

```

5.2.8. Масиви като фиктивни аргументи на функции

Функциите могат да имат фиктивни аргументи, които са масиви. Те се описват по два начина, както е показано в следното примерно заглавие на функция

```
float skal_pro ( int n, float x[], float* y).
```

Масивите винаги се предават по адрес (чрез указател), т.е. името на масива е указател към първия му елемент.

Ако в друга функция се съдържа операторът

```
float z=scal_pro (10, a, b);
```

то x ще получи стойност адреса на масива a, а y – адреса на масива b.

а. Програми с функции, обработващи масиви.

Програма 5.10. Програмата разменя стойностите на елементите на два масива чрез функция.

```

#include <stdio.h>
//Дефиниция на функция за размяна на стойностите на
елементите на два масива
void swap_arr ( int n, int x[], int* y )
{
    int buf;
    for ( int i=0; i<n; i++ ){
        buf  = x[i];
        x[i] = y[i];
        y[i] = buf;
    }
}
void main() {
    const BrEl = 5; //Макс. дължина на a и b
    int a[BrEl], b[BrEl];
    for ( int i = 0; i < BrEl; i++ ){
        a[i] = i; b[i] = BrEl-1-i;
    }
}

```

```

printf("ПРЕДИ РАЗМЯНАТА:\n");
for ( i = 0; i < BrEl; i++ ) {
    printf("a[%d]=%d  ", i, a[i]);
    printf("b[%d]=%d\n", i, b[i]);
}
swap_arr(BrEl, a, b );
printf("СЛЕД РАЗМЯНАТА:\n");
for ( i = 0; i < BrEl; i++ ) {
    printf("a[%d]=%d  ", i, a[i]);
    printf("b[%d]=%d\n", i, b[i]);
}
}

```

Програма 5.11. Програма с функция сумиране на елементите на едномерен масив

```

#include <stdio.h>
// Функция за сумиране на едномерен масив
int SumMas1 (int n, int x[] )
{
    for (int i=0,s=0;i<n;i++) s+=x[i];
    return s;
}
// Главна функция main
void main()
{
    int X[]={1, 2, 3, 4, 5, 6, 7, 8, 9};
    printf("Сума на всички елементи на X: %d\n",
        SumMas1(9,X));
    printf("Сума на първите 5 елемента на X: %d\n",
        SumMas1(5,X));
    printf("Сума на 3-ия,4-ия,5-ия и 6-ия елементи: %d\n",
        SumMas1(4,X+3));
}

```

Програма 5.12. Използва функции за въвеждане и сумиране на елементите на двумерен масив по редове

```

#include <stdio.h>
// Функция за въвеждане на едномерен масив
void ReadMas1 (int &n, int x[])
{
    if (n==0) {
        printf("Въведете броя на елементите: ");
        scanf("%d", &n);
    }
    for (int i=0;i<n;i++)

```

```

        {printf("%d-и елемент:", i+1); scanf("%d", &x[i]);}
    }
// Функция за сумиране на едномерен масив
int SumMas1 (int n, int x[])
{
    for (int i=0, s=0; i<n; i++) s+=x[i];
    return s;
}
// Главна функция
void main()
{
    int r[10][12], //Двумерен масив
        suma[10], //Масив на сумите
        BrRed,    //Брой на редовете
        BrCol;    //Брой на колоните
    printf("\nBrRed="); scanf("%d", &BrRed);
    BrCol=0;
    for (int i=0; i<BrRed; i++)
    {
        //Въвеждане на i-тия ред на масива r
        ReadMas1(BrCol, r[i]);
        puts("Елементи:");
        //Извеждане на i-тия ред на масива r
        for (int j=0; j<BrCol; j++) printf("%d ", r[i][j]);
        //Сумиране на i-тия ред на масива r
        suma[i]=SumMas1( BrCol, r[i] );
        //Извеждане на сумата на i-тия ред на масива r
        printf("Сума: %d\n", suma[i]);
    }
}

```

Програма 5.13. Програма, която въвежда два вектора и намира скаларното им произведение

```

#include <stdio.h>
// Функция за въвеждане на едномерен масив
void ReadMas1 (int &n, float x[])
{
    if (n==0) {
        printf("Въведете броя на елементите: ");
        scanf("%d", &n);
    }
    for (int i=0; i<n; i++)
        {printf("%d-и елемент:", i+1); scanf("%f", &x[i]);}
}

// Функция за скаларното произведение на два вектора

```

```

float ScalPro (int n, float x[], float y[])
{
    float s=0;
    for (int i=0; i<n; i++) s+=x[i]*y[i];
    return s;
}
// Главната функция main
void main()
{
    float a[20], b[20]; //2 вектора
    int BrEl=0;
    ReadMas1(BrEl, a); ReadMas1(BrEl, b);
    printf("Скаларното произведение е %f\n",
        ScalPro(BrEl, a, b));
}

```

Програма 5.14. Програма, която въвежда матрица и вектор и намира произведението им

```

#include <stdio.h>
// Функция за въвеждане на едномерен масив
void ReadMas1 (int n, float x[])
{
    for (int i=0;i<n;i++)
        {printf("%d-и елемент:",i+1);scanf("%f",&x[i]);}
}

// Функция за скаларното произведение на два вектора
float ScalPro (int n, float x[], float* y)
{
    float s=0;
    for (int i=0; i<n; i++)
        s+=x[i]*y[i];
    return s;
}
// Функция за умножение на матрица по вектор
void MatrVect (int M, int N, float a[][5],
                float b[], float p[])
{
    for ( int i=0; i<M; i++) p[i]=ScalPro(N,a[i],b);
}

void main()
{
    int BrRed, BrSt;
    float A[5][5], B[5], P[5];
    printf("BrRed="); scanf("%d",&BrRed);
}

```

```

printf("BrSt="); scanf("%d",&BrSt);
for ( int i=0; i<BrRed; i++) ReadMas1(BrSt, A[i]);
ReadMas1(BrSt, B);
MatrVect(BrRed, BrSt, A, B, P);
for ( i=0; i<BrRed; i++)
    printf("%f ",P[i]);
printf("\n");
}

```

б. Програми с функции, обработващи низове.

Програма 5.15. Програма с функция за копиране на част от низ в нов низ

```

#include <stdio.h>
//Функция за копиране на група символи от низ А в низ В
void cpypartstr (char a[], char b[], int n, int br)
{
    for (int i=0;i<br;i++) b[i]=a[n-1+i];
    b[i]='\0';
}
// Главната функция
void main()
{
    char A[]="Библиография", B[11];
    cpypartstr(A, B, 7, 4);
    puts(B);
}

```

Програма 5.16. Намиране на максимален елемент в масив от символни низове с функции

```

#include <stdio.h>
#include <string.h>
// Функция за въвеждане на едномерен масив
void ReadMasNiz(int &n, char x[][11])
{
    if (n==0) {printf("n="); scanf("%d",&n);getchar();}
    for (int i=0;i<n;i++)
        {printf("%d-и елемент:",i+1);scanf("%s",&x[i]);}
}
// Функция за намиране на индекса на максималния елемент
int NomMaxElNiz(int n, char x[][11])
{
    int m=0;
    for (int i=1;i<n;i++)
        if (strcmp(x[i],x[m])>0) m=i;
    return m;
}

```

```

// Главна функция
void main()
{
    int BrEl;
    char MasNiz[10][11];
    printf("BrEl="); scanf("%d",&BrEl);
    ReadMasNiz(BrEl, MasNiz);
    //Намиране индекса на макс. елемент
    int n=NomMaxElNiz(BrEl, MasNiz);
    printf("Макс. елемент: &s\n",MasNiz[n]);
}

```

Програма 5.17. Намиране на максимален елемент във всеки ред на двумерен масив от низове

```

#include <stdio.h>
#include <string.h>
// Функция за въвеждане на едномерен масив
void ReadMasNiz(int &n, char x[][11])
{
    if (n==0) {printf("n="); scanf("%d",&n);getchar();}
    for (int i=0;i<n;i++)
        {printf("%d-и елемент:",i+1);scanf("%s",&x[i]);}
}
// Функция за намиране на индекса на максималния елемент
int NomMaxElNiz(int n, char x[][11])
{
    int m=0;
    for (int i=1;i<n;i++)
        if (strcmp(x[i],x[m])>0) m=i;
    return m;
}
// Главна функция
void main()
{
    int BrRed, BrCol=0, n;
    char MasNiz2[10][15][11]; int IndMax[10];
    printf("BrRed="); scanf("%d",&BrRed);
    for (int i=0; i<BrRed; i++) {
        cout<<i<<"-и ред:"<<endl;
        ReadMasNiz(BrCol, MasNiz2[i]);
        IndMax[i]=NomMaxElNiz(BrCol, MasNiz2[i]);
        n=IndMax[i];
        printf("Макс. елемент: %d %s\n",n,MasNiz2[i][n]);
    }
}

```

5.2.9. Функции с подразбиращи се аргументи

Функциите в C++ могат да имат подразбиращи се аргументи, т.е. аргументи, на които са зададени подразбиращи се стойности. За задаване на подразбиращи се стойности на аргументите се използва знака за присвояване. Ето една примерна програма:

Програма 5.18. Функция с подразбиращи се аргументи

```
#include <stdio.h>
void func(int x, float y=2.25, char* s="Информатика")
{
    printf("x=%d, y=%f, s=%s\n", x, y, s);
}

void main()
{
    func(5);
    func(5, 3.75);
    func(5, 4.75, "Математика");
}
```

Аргументите `y` и `s` на функцията `func` са подразбиращи се и техните подразбиращи се стойности са съответно `2.25` и `"Информатика"`. Тези стойности ще бъдат предадени на функцията като аргументи, ако при нейното извикване не бъдат зададени други стойности на подразбиращите се аргументи. Функцията `func` може да бъде извикана по три различни начина (с един, с два и с три аргумента), тъй като има два подразбиращи се аргумента. Например,

Резултатите от тези три изпълнения на функцията `func` ще бъдат съответно:

```
x=5, y=2.25, s=Информатика
x=5, y=3.75, s=Информатика
x=5, y=4.75, s=Математика
```

При използване на функции с подразбиращи се аргументи трябва да се има предвид следното правило: Ако един аргумент на дадена функция е подразбиращ се, то всички аргументи, които се намират след него в списъка на аргументите също трябва да бъдат подразбиращи се. Следователно дефиниция от вида:

```
int f ( int x, float y = 1.2, char *s ) //ГРЕШКА!
{
    .....
}
```

е некоректна, тъй като вторият параметър `y` е подразбиращ се, а аргументът `s`, който се намира след него в списъка на фиктивните аргументи, не е подразбиращ се.

Програма 5.19. Програма с функция за копиране на част от низ в нов низ. Ако често се налага да извличаме от ЕГН-тата на различни лица датата на раждане, програма 5.18 можем да преобразуваме в следния вид:

```
#include <stdio.h>
//Функция за копиране на група символи от низ А в низ В
void cpypartstr (char a[], char b[], int n, int br=2)
{
    for (int i=0;i<br;i++) b[i]=a[n-1+i];
    b[i]='\0';
}
void main()
{
    char A[]="Библиография", EGN[11]="8509154265",
        B[11], D[3];
//На низа В ще се присвои "граф"
    cpypartstr(A, B, 7, 4);
    puts(B);
//На низа D ще се присвои "15"
    cpypartstr(EGN, D, 5);
    puts(D);
}
```

5.2.10. Предефинирани функции

В програма на C++ могат да се дефинират няколко функции с едно и също име. Такива функции се наричат предефинирани (overloading functions). Те обаче трябва да се различават по брой аргументите или по типа поне на един от своите аргументи. Те са подходящи за реализация на еднакви по смисъл операции, които се прилагат върху различни по тип данни или върху различен брой обекти, както е показано в следващата програма.

Програма 5.20. Предефинирани функции

```
#include <stdio.h>
//Дефиниция на функция max с два параметъра
int max(int x,int y) // Връща по-голямото от две числа
{
    if (x>y) return x;
    else return y;
}
//Дефиниция на функция max с три параметъра
//Връща най-голямото от три числа
int max(int x,int y,int z)
{
```

```

    if (x>y)
        if (x>z) return x;
            else return z;
    else if (y>z) return y;
        else return z;
}
void main ()
{
    int X,Y,Z;
    printf("X="); scanf("%d",&X);
    printf("Y="); scanf("%d",&Y);
    printf("Z="); scanf("%d",&Z);
// Извикване на първата функция
    printf("%d\n",max(X,Y));
    printf("%d\n",max(Y,Z));
    printf("%d\n",max(Z,X));
// Извикване на втората функция
    printf("%d\n",max(X,Y,Z));
}

```

5.2.11. Вградени функции

Едно от предимствата на функциите се състои в това, че техният код се съхранява само на едно място в паметта, независимо от броя на изпълненията им в дадена програма. По този начин силно се намалява разходът на памет. Но икономията на памет е за сметка на бързодействието, тъй като всяко обръщение към дадена функция е свързано с извършването на някои служебни операции (запомняне на възвратния адрес, предаване на аргументите и преход към стартовия адрес на дадената функция и др.), които отнемат време. В конвенционални програми подобно забавяне с без значение, но има програми, в които е важно да се постигне максимално бързодействие. Примери за такива програми са програмите за управление на различни устройства и технологични процеси. Програми от този вид работят в режим на реално време, което означава, че работата им трябва да бъде съобразена с различни временни ограничения, зависещи от управляваните обекти.

За повишаване бързодействието на програмите, C++ предлага възможност да дефинираме вградени функции (inline functions). Те се дефинират и използват както всички останали функции с тази разлика, че ползват модификатор inline, т.е. дефинициите им започват с ключовата дума inline. Ето един пример:

```

inline void bell(void){
    cout << '\a';
}

```

Изпълнението на тази функция предизвиква издаване на звуков сигнал от компютъра.

Кодът на вградената функция се копира на всяко място в програмата, където има обръщение към нея. По този начин броят на служебните операции, които трябва да се изпълняват при извикване на вградените функции намалява, а това води до увеличаване на бързодействието. Препоръчително е вградените функции да бъдат "къси". В противен случай разходът на памет може да се увеличи твърде много, тъй като техният код се копира многократно.

5.2.12. Статични променливи във функциите

Статичните променливи и константи се дефинират във функциите чрез ключовата дума `static` и се наричат статични вътрешни променливи и константи. Дефинирането трябва да е съпроводено с инициализация. Те са достъпни само в рамките на функциите, в които са дефинирани, но се създават само веднъж при стартиране на програмите и съществуват през цялото време на тяхното изпълнение. Накратко, статичните вътрешни променливи са локална постоянна памет на функциите, в които са дефинирани.

Основно приложение на статичните вътрешни променливи е съхранението на данни (стойности), които се получават в резултат от изпълнението на дадена функция и се използват от същата функция при следващо нейно изпълнение.

Програма 5.21. Програма с функция със статична променлива

```
#include <stdio.h>
float dobavi(float x)
{
//Дефиниране и инициализация на статична променлива
    static float s=0;
    return s+=x;
}
void main()
{
    float a[5]={ 1, 2, 3, 4, 5 }, suma;
    for (int i=0; i<5; i++) suma=dobavi(a[i]);
    printf("Сумата на числата е %f\n",suma);
}
```

5.2.13. Функции, които връщат указатели

Върнатите стойности на функциите могат да бъдат от произволен тип, включително и указатели. Ето как изглежда дефиницията на една функция с име `f`, която връща указател към тип с име `type`:

```
type *f(.....)
{
```

```

. . . . .
return <указател към type>;
}

```

5.2.14. Указатели към функции

Указателите към функции са указатели, чиито стойности не са адреси на данни, а начални адреси на функции. Ето как изглежда дефиницията на указател към функция:

```
float (*fp)(int, float);
```

Тази дефиниция означава, че `fp` е указател към функция, която има два аргумента от тип `int` и `float` и връща стойност от тип `float`. Скобите, ограждащи указателя `fp`, са задължителни. Ако те липсват, дефиницията ще означава, че `fp` е функция, която връща указател към `float`.

На указателите към функции могат да се присвояват началните адреси на различни функции, след което извикването на функциите може да става чрез указателите. Началните адреси на функциите се означават чрез имената на функциите, без да се поставят скоби след тях и без да се задават аргументи. Например, нека е дефинирана една функция с име `f1`:

```
float f1(int x, float y)
{
. . . . .
}

```

Началният адрес на функцията `f1` може да бъде присвоен на указателя `fp` чрез израза:

```
fp=f1;
```

От този момент, функцията `f1` може да бъде извиквана чрез указателя `fp` по следния начин:

```
(*fp)(5, 3.4);
```

Някои компилатори, в това число и BORLAND C++, допускат извикването на функция чрез указател да става само чрез името на указателя, а не чрез съдържанието на указателя, поставено в скоби. Имайки предвид това, вместо `(*fp)(5, 3.4)`, може да се използва по-простият запис `fp(5, 3.4)`. Чрез указателя `fp` могат да бъдат извиквани и други функции, като преди това трябва да му бъдат присвоявани техните начални адреси.

Програма 5.22. Указател към функция

```

#include <iostream.h>
#include <math.h>
//Дефиниране на функция
float f1(int x, float y)
{
return x+y;
}

```

```

}
//главна ф-ия
void main()
{
    //Дефиниране на указател към функция
    float (*p)(int, float);
    p=f1; //Инициализация на указателя
//обръщение към ф-та чрез указателя към нея
    printf("%f\n", (*p)(5, 3.4));
}

```

Указателите към функции намират приложение предимно в два случая. Първият случай е свързан с разработването на драйвери за различни устройства, с които работи дадена програма. Драйверите се оформят като отделни функции, а използването им от програмата става чрез указатели към функции. По този начин се постига независимост на програмата от конкретните драйвери в даден момент. Освен това добавянето на нови драйвери не изисква промени в програмите, които ги използват. Вторият, често срещан случай на използване на указатели към функции, е свързан с предаването на функции като аргументи на други функции. За да може една функция да се предаде като аргумент на друга функция, то съответният фиктивен аргумент трябва да бъде указател към функция. Програма 5.25 илюстрира този случай.

Програма 5.23. Табулация на функция. Предаване на функция като аргумент.

```

#include <stdio.h>
#include <math.h>
//Дефиниция на функция tabul
void tabul (double (*f)(double),float y[],float x[],
            int n,float st=1)
{
    for (int i=0; i<n; i++) y[i]=(*f)(x[i]=(i+1)*st);
}
//Дефиниция на функция f1
double f1(double x)
{
    return x*x+2;
}
//Дефиниция на функция f2
double f2(double x)
{
    return x*x-1;
}
//Дефиниция на функция main

```

```

void main() {
    const br=5;    //Максимална дължина на масивите
    int i;
    float x[br], y[br];
    tabul(f1,y,x,br,2);
    printf("\n x f1(x)\n");
    for (i=0; i<br; i++) printf("%4.1f %f\n",x[i],y[i]);
    tabul(f2,y,x,br);
    printf("\n x f2(x)\n");
    for (i=0; i<br; i++) printf("%4.1f %f\n",x[i],y[i]);
    tabul(sin,y,x,br,0.1);
    printf("\n x sin(x)\n");
    for (i=0; i<br; i++) printf("%4.2f %f\n",x[i],y[i]);
}

```

Функцията `tabul` табулира функция (математическа) на един аргумент. Функцията, която се табулира, се предава като първи аргумент на функцията `tabul`. След изпълнение на функцията `tabul`, масивите `y` и `x` съдържат съответно стойностите на функцията, която се табулира (предава се като първи аргумент на функцията `tabul`) и стойностите на аргумента, за които е изчислена функцията. Аргументът `n` съдържа броя на стойностите в масивите `y` и `x`. В програма 5.25 масивите са дефинирани във функцията `main`, като дължината им е зададена чрез константата `br`. Аргументът `st` задава стъпката на изменение на стойностите на аргумента на функцията, която се табулира. Аргументът `st` има подразбираща се стойност единица. Следователно, ако при извикване на функцията `tabul` аргументът `st` не бъде зададен, то неговата стойност ще бъде 1. Във функцията `main`, функцията `tabul` се извиква три пъти, за да се табулират функциите с имена `f1`, `f2` и `sin`. Функциите `f1` и `f2` съответствуват на две математически функции, които имат следните аналитични записи: $f1(x) = x*x+2$ и $f2(x) = x*x-1$. Функцията `sin` е вградената функция синус. При първото извикване на функцията `tabul` аргументът `st` има стойност 2. При второто извикване аргументът `st` не се предава. Следователно функцията `f2` ще бъде табулирана със стъпка 1, колкото е подразбиращата се стойност на аргумента `st`.

Някои компилатори, в това число и BORLAND C++, допускат опростено деклариране на указателите към функции като фиктивни аргументи на функциите. Опростяването се състои в премахване на скобите и `*` пред имената на указателите към функции. Следвайки това правило, функцията `tabul` може да бъде дефинирана и по този начин:

```

void tabul (double f(double),float y[],float x[],
            int n,float st=1)
{
    for (int i=0; i<n; i++) y[i]=f(x[i]=i*st);
}

```

5.3. ВЪНШНИ ПРОМЕНЛИВИ

5.3.1. Дефиниране и използване на външни променливи

В програмите на C++ могат да се дефинират обекти (променливи, константи и масиви) извън тялото на която и да е от функциите на съставната програма. Такива обекти се наричат **външни**. Те не се създават в програмния стек, а в друга област от паметта, наречена статична памет. При това се създават еднократно при стартиране на програмата и съществуват през цялото време на нейното изпълнение.

Областта на действие на една външна променлива се простира от мястото на нейната дефиниция до края на файла. Това означава, че достъп до дадена външна променлива ще имат само функциите, чиито дефиниции са разположени след дефиницията на външната променлива.

Външните променливи се използват основно в два случая:

1. Когато функцията е с голям брой аргументи. Целесъобразно е част от тях да са външни. Така програмата става по-прегледна и по бърза.

2. Когато две или повече функции работят с общи данни, но нямат обръщания една към друга.

Програма 5.24. Дефиниране и видимост на външни променливи

```
#include <stdio.h>
int x = 1; //Дефиниция на външна променлива x
void f1()
{
    x++; printf("f1: x=%d\n", x);
}
int y = 0; //Дефиниция на външна променлива y
void f2()
{
    y += x;
    printf("f2: x=%d y=%d\n", x, y);
}
int z = 2; //Дефиниция на външна променлива z
void f3()
{
    z = x + y;
    printf("f3: x=%d y=%d z=%d\n", x, y, z);
}
void main()
{
    f1(); f2(); f3();
}
```

В тази програма са дефинирани три външни променливи - x, y и z. Дефиницията на променливата x се намира преди функциите main, f1, f2 и f3, поради което и всички тези функции имат достъп до нея. Променливата y е достъпна за функциите f2, f3 и main, но не и за функцията f1, тъй като дефиницията на функцията f1 е разположена преди дефиницията на променливата y. И накрая, променливата z е достъпна само за функциите f3 и main. Резултатът от изпълнението на програма 5.12 ще бъде следния:

```
f1: x = 2
f2: x = 2   y = 2
f3: x = 2   y = 2   z = 4
```

5.3.2. Външни и вътрешни променливи с еднакви имена

Езиците C++ и C допускат наличието на външни и вътрешни променливи с еднакви имена. Ако в една функция са дефинирани вътрешни променливи, чиито имена съвпадат с имената на външни променливи, функцията няма да има пряк достъп до външните променливи, тъй като имената на вътрешните променливи "засенчват" имената на външните променливи. В езика C++ (но не и в езика C) този проблем се решава чрез използване на оператора ::, който се нарича **оператор за принадлежност**. Поставянето на оператора :: пред името на дадена променлива означава външна променлива с указаното име. Като пример за използване на оператора :: за разграничаване на външни и вътрешни променливи с еднакви имена с показана програма 5.13.

Програма 5.25. Автоматични и външни променливи с еднакви имена. Използуване на оператора ::

```
#include <stdio.h>
int x=10, y=14, z=7;          //Външни променливи
//Дефиниция на функция suma
void suma(int x, int y, int &z)
{
    z=x+y; //Автоматични променливи
    puts("Изведено от функцията suma:");
    printf("x=%d\n", x);
    printf("y=%d\n", y);
    printf("z=%d\n", z);
    printf("::x=%d\n", ::x);
    printf("::y=%d\n", ::y);
}
//Дефиниция на функция main
void main()
{
    int x=2, y=3;           //Автоматични променливи
```

```
x+=y;
y>::y+ 5;
puts("Изведено от функцията main:");
printf("x=%d\n", x);
printf("y=%d\n", y);
printf("::x=%d\n", ::x);
printf("::y=%d\n", ::y);
suma(::x, y, ::z);
puts("Изведено от функцията main:");
printf("::z=%d\n", ::z);
}
```