

Лекция 8.

Обработка на данните чрез
групиране. Подпрограми

*Можем ли в програмите да
използваме предимствата на
обобщението?*

Подпрограми: АМ

Подпрограмата е първата по-обща абстракция по отношение на обработката на данните въведена в ЕП

- **Същност:** Подпрограмите са средство за отделяне и идентифициране на подалгоритъм от даден алгоритъм при програмно описание на този алгоритъм
- **Предназначение:**
 - **групиране на съвкупност от действия**, които са част от даден алгоритъм, обикновено логически самостоятелна част (подалгоритъм). В повечето случаи на тази съвкупност от действия се съпоставя име – **име на подпрограмата**
 - **указване изпълнението** на цялата съвкупност от действия само чрез съпоставеното име



Ролята на подпрограмата при компютърно моделиране на алгоритми може да се сравни с ролята на обобщението и въвеждането на нови понятия в процеса на натрупване на човешки знания

Подпрограми: Видове

В ЕП се срещат **два вида** подпрограми:

- **Функции** — служат за описание на подалгоритми, чието множество от изходни данни се състои от един елемент, който се нарича *резултат на функцията*
- **Процедури** — използват се за описание на произволни подалгоритми

 В C++ има само подпрограми-функции

Подпрограми: Взаимодействие между различните части на програмите – механизъм на предаване на параметри

- **Видове взаимодействие** за определяне на входните и изходните данни при подпрограмите:
 1. с **външната** среда (потребителя и периферните устройства за вход и изход)
 2. с **вътрешната** среда (с други части на програмата)
- Средство осигуряващо втория вид взаимодействие: "*механизъм на предаване на параметри*"

Механизмът на предаване на параметри осигурява многократното използване на подпрограмите (свойството масовост), давайки възможност подпрограмите да се изпълняват за различни множества от входни данни и да се получават различни изходни резултати

Подпрограми: Механизъм на предаване на параметри

Същността на механизъм на предаване на параметри се състои в това, че задаването на множествата от входни и изходни данни на подпрограмите се разделя на два етапа:

- При описание на съвкупността от действия (деклариране и дефиниране), които подпрограмата групира се използват т.нар. **формални параметри** – служат за определяне на множеството от входни и изходни данни на подпрограмата само в най-общ вид (брой и тип), като им съпоставят формални (условни) имена (без да определят конкретните им стойности)
 - 🔊 Поради специфичното предназначение на функциите, при тях обикновено множеството от изходните данни се състои от един елемент и неговият тип определя типа на резултата на самата функция. Така че при функциите списъкът формални параметри в общия случай определя само множеството от входни данни
- При активирането (изпълнение) на подпрограмата се използват т.нар. **фактически параметри**, които задават конкретните стойности на входните и изходните данни
 - 🔊 Възможността формалните параметри на подпрограмите да се заместват с различни фактически параметри при всяко извикване на подпрограмата се нарича *механизъм на предаване на параметри* (аналогично на дефиниране и намиране на стойност на функция в математиката)

Подпрограми: Предимства

- *Лаконичност* при програмна реализация на алгоритмите, тъй като там където е необходимо да се изпълнява цялата група от действия, е достатъчно да се укаже само името на групиращата подпрограма, без да се налага тази последователност от действия отново да се задава
- *Повишаване надеждността на програмите*, тъй като алгоритъма на задачата се разбива на малки части (подпрограми), които самостоятелно се проектират, програмират, тестват и настройват и по този начин вероятността да се допусне грешка е значително по-малка
- *Многократно използване* на едни и същи подпрограми като елементи при разработване на различни програми. Това от своя страна отново води до повишаване надеждността на програмите, в следствие на възможността за използване на предварително готови и проверени програмни части

Функции в C++: Особенности на реализацията

Реализацията на подпрограмите-функции в C++ следва общата идея на тази абстракция, но има по-широко семантично тълкувание:

Могат да се разглеждат като тип данни, което дава възможност изборът на функцията, която ще бъде изпълнена да се отложи за момента на изпълнение на програмата. Използват се указатели към типа функция

Функции в C++: ЗМ – Синтаксис и семантика за деклариране и дефиниране

Синтаксисът се състои от заглавие и блок ({ . . . }):

```
<тип на резултата>      незад  <име> ( <списък формални параметри>      незад )  
{  
    <декларация на локални променливи>      незад  
    <оператори>      незад  
    return <израз>;      незад  
}
```

- Ако <тип на резултата> е пропуснат се подразбира `int`. Ако не той може да бъде:
 - всички елементарни типове: `int`, `char`, `float`, и т.н.
 - `void` – функцията не връща резултат
 - указатели (напр. указател към масив)
 - структури
- <име> е името на функцията
- <оператори> описват алгоритъма на функцията
- `return <израз>` служи за връщане на управлението на програмата в точката на извикване на функцията и заедно с това връща нейния резултат, т.е. стойността на <израз> (<израз> е от тип <тип на резултата> или може да се преобразува към него). Тази част може да се пропусне ако ф-ята връща `void`



С този синтаксис, функции не могат да се декларират и дефинират като част от други функции

Функции в C++: ЗМ – Синтаксис на списъка формални параметри

Синтаксисът допуска 3 основни форми:

- (1) празен списък или `void` – функцията няма входни данни
- (2) $\langle \text{тип}_1 \rangle \langle \text{име}_1 \rangle = \langle \text{израз}_1 \rangle_{\text{незад}}$, $\langle \text{тип}_2 \rangle \langle \text{име}_2 \rangle = \langle \text{израз}_2 \rangle_{\text{незад}}$ и т.н.
– функцията има определен брой параметри
- (3) $\langle \text{тип}_1 \rangle \langle \text{име}_1 \rangle = \langle \text{израз}_1 \rangle_{\text{незад}}$, . . .
– т.е. последният параметър е заместен с . . . ,
тогава функцията има променлив брой параметри. Може да се използва ако дефинираме универсална функция, за която и броя и типа на параметрите се променя (напр. като `printf`)
 - $\langle \text{тип}_i \rangle \langle \text{име}_i \rangle = \langle \text{израз}_i \rangle_{\text{незад}}$ задава i -ят параметър на ф-ята (съответно – тип, име и евентуално стойност по подразбиране)
 - $\langle \text{тип}_i \rangle$ може да бъде всеки елементарен тип (`int`, `char`, `float`, ...), указател, псевдоним, структура, обединение, клас, масив, функция
 - параметрите със стойност по подразбиране се пишат в края на списъка

Функции в C++: ЗМ – Семантика на списъка

формални параметри

- **Обща семантика:** Задава броя и типа на входните, входно-изходните, и изходни данни на функцията и им съпоставя формални имена
- **Семантика според вида на параметъра:**
 - **Формални параметри, които се предават по адрес** – това са параметрите от тип *псевдоним* (&) или *указател* (*). Те служат главно за деклариране на *изходни* или *входно-изходни данни*, въпреки че могат да се използват и за деклариране на входни данни
 - 🔊 За предаване на изходни или входно-изходни параметри по-удобен и по-съвременен е начинът чрез псевдоними
 - **Формални параметри, които се предава по стойност** – това са параметрите, от всички останали типове. Те служат за деклариране само на *входни данни*
 - **Формални параметри-константи** – тези пред типа, на които има служебната дума `const`. Най-често като **параметри-константи** се декларираат някои **параметри** от тип *псевдоним* с цел да се използват предимствата на този начин на предаване на параметри (вж. Слайд 10), като в същото време се забрани модификацията им в блока на функцията

Функции в C++: ЗМ – Примери за деклариране и дефиниране

- **Параметри по стойност:**

```
float addf(float a, float b)
{ float c=a+b; a = a + 10; return c; }
```

- **Параметри по адрес:**

(1) чрез *псевдоним* (&):

```
void swap(int& a,int& b){
    int temp=a;
    a=b; b=temp;
}
```

(1) чрез *указател* (*):

```
void swapP(int* pa,int* pb){
    int temp=*pa;
    *pa=*pb; *pb=temp;
}
```

- **Параметри със стойност по подразбиране:**

```
int init(int count=0, int start=1, int end=100) {
    //...;
    return count;
}
```

- **Параметри-константи:**

```
void PrintInt( const int &IntValue ) {
    printf( "%d\n", IntValue );
}
```



Ако самият указател е изходен или входно-изходен параметър, то се декларира псевдоним на указател (* &) или указател към указател (**)

Функции в C++: ЗМ – Допълнителни изисквания за блока на функциите

- Тъй като блокът на всяка функция е самостоятелен, то имената на формалните параметри, както и на променливите, константите и т.н. всички програмни елементи, декларирани в блока на функцията е необходимо да бъдат уникални само в рамките на този блок (а не в рамките на цялата програма) Те се наричат *локални*
- При това важи едно ограничение: Ако някое локално име съвпада с глобално, то съответния глобален обект е недостъпен в рамките на тази функция

```
const int c=2;                                //глобално c
void fgl(){cout<<endl<< "global: "<<c;};
void main() {
    int c=4;                                    //локално c
    cout<<endl<< "local: "<<c;
    fgl();
}
```
- На всички изходни параметри задължително трябва да бъдат присвоени стойности в блока на функцията. Ако функцията връща резултат различен от `void`, то блокът трябва задължително да включва оператор `return`

Функции в C++: ЗМ – прототипи на функции

Декларация на функция, която е направена преди и отделно от дефиницията на функцията се нарича **прототипи на функцията (декларация)**

- **Синтаксис** – състои се само от заглавие на функция (вж слайд 7 и 8):

<тип на резултата> <име> (<списък формални параметри> незад) ;


, допуска се имената на формалните параметри да се пропуснат (не е препоръч.)


- **Пример:**


```
int add(int a, int b); //!!! ";" в края на декларацията
int GetIntegerInRange(int low, int high);
```

- **Приложение:**

- Прототипите се използват за инициализация на указатели към функции, преди тези функции да са дефинирани
- Списъкът формални параметри на **прототипа** се използва за проверка на съответствието м/у аргументите от дефиницията и активирането на ф-ята

 Обща практика е всички прототипи на функции да се включват в един заглавен файл (напр. mylib.h) . Тогава трябва да включим този заглавен файл във всички програмни файлове, които го използват (#include "mylib.h")

 Няма специални изисквания относно реда на деклариране и дефиниране на функциите, дори прототипите могат да се декларират в блока на функцията-клиент

 Стойностите по подразбиране се задават или само в прототипа или само в дефиницията

Функции в C++: ЗМ – Оператор за активиране на функции

- **Синтаксис** – той е оператор-израз:

`<име> (<списък фактически параметри> незад)`

- `<име>` е име на предварително декларирана функция
- `<списък фактически параметри>` е последователност от изрази (фактическите параметри, т.е. тези които се подават на ф-ята), разделени с “,” и трябва донякъде да съответстват по реда си, по броя и типа на формалните параметри от описанието
- ако съответните формални параметри имат стойности по подразбиране, които съвпадат с тези на фактическите, то последните могат да се пропуснат, но само ако са в края на списъка

- **Семантика:**

Извикването на ф-ята има типа на резултата на ф-ята и предизвиква нейното изпълнение, като конкретните стойности на входните и изходните данни се определят от зададените при това извикване фактически параметри

Функции в C++: ЗМ – Примери за активиране на функции

Активиране на функциите дефинирани на слайд 10:

- **Параметри по стойност:**

```
float x=10, y=20, z=0;
```

```
z = addf(x, y);
```

```
cout<< x <<" , "<< y <<" , "<< z << endl; //изв. 10, 20, 30
```

- **Параметри по адрес:**

(1) чрез *псевдоним* (&):

```
float x=10, y=20;
```

```
swap(x, y);
```

```
cout<< x <<" , "<< y << endl;
```

```
//ще изведе 20, 10
```

(1) чрез *указател* (*):

```
float x=10, y=20;
```

```
swapP(&x, &y);
```

```
cout<< x <<" , "<< y << endl;
```

```
//ще изведе 20, 10
```

- **Параметри със стойност по подразбиране:**

```
int i = init(100); //правилно init(100,1,100)
```

```
int j = init(100,,50); //!!!неправилно
```


Функции в C++: ЗМ – сравнение между видовете формалните параметри

Формални параметри, които се предават по адрес (ФПА)

Формални параметри, които се предава по стойност (ФПС)

1. Синтактичните особености:

- а) **ФПА** се декларира като псевдоними или указатели
- б) Фактическите параметри съответни на **ФПС** могат да бъдат произволни изрази, докато тези съответни на **ФПА** – само променливи от съвместим тип

2. Семантични особености:

- а) **ФПА** могат да служат като изходни и като входно-изходни, докато **ФПС** служат само за задаване на входни данни
- б) Измененията, които ф-ята извършва върху **ФПС** не се отразяват върху съответните им фактически параметри, докато при **ФПА** се отразяват

3. Начина на реализацията им в езика C++ (защо така се наричат?):

- а) За фактическите параметри съответни на **ФПС** при извикване на ф-ята се заделя памет в стека. Измененията, които ф-ята извършва с техните стойности се записват в тази памет. След приключване на изпълнението, тази памет се освобождава
- б) За фактическите параметри съответни на **ФПА** в стековата памет се записва само адреса на съответния фактически параметър. Ето защо измененията, които ф-ята извършва с техните стойности се записват в първоначално заделената за тях памет



Препоръчва се, ако формалните параметри са от някой тип, който заема много памет, те да се декларира като **ФПА** (дори и да са входни), или като параметри-константи

Функции в C++: ЗМ – Особенности

- В C++ е позволено да се декларират функции с еднакви имена, но с различен списък формални параметри (не се допуска разлика само в типа на резултата), т.е. т.нар. **натоварване на имената на функциите** (*overloading*):

```
float add(float a, float b)
{ return a + b; }
```

```
int add(int a, int b)
{ return a + b; }
```

...

```
int i,j,k;
```

```
float x,y,z;
```

```
z = add(x,y); // float add(float a, float b); се извиква
```

```
k = add(i,j); // float add(int a, int b); се извиква
```

- В C++ с цел ефективност при изпълнение могат да се декларират т.нар. **inline функции**:

```
inline int add(int a, int b)
{ return a + b; }
```

, което изисква от компилатора извикването на `add(.,.)` да бъде заместено със самия блок на ф-ята, т.е. `a + b` се вмъква (не винаги) в точката на извикването

Функции в C++: Масивите като параметри на функции

Ако параметър на функция е масив, то:

- се допуска да не се зададе размерът по първото измерение на масива.

Компилаторът ще запълни масива според размера на фактическия параметър:

```
int getline(char s[]) {  
    ...  
    s[i]=c; // or *(s+i)= c;  
    ...  
}
```

В такъв случай е добре като допълнителен параметър да се подаде пропуснатия размер, за да може по-лесно и сигурно да се обработва масива:

```
void print(int a[], int n)  
/* Отпечатва елементите на масив */  
{  
    int i;  
    for (i = 0; i < n; i++)  
        cout << a[i] << " ";  
    cout << "\n";  
}
```

- C++ предава масивите винаги по адрес, независимо че това не е зададено (чрез псевдоним)