ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ – СОФИЯ

ИНФОРМАТИКА

част втора

Основи на програмирането на C++

лектор: гл. ас. д-р Стефан М. Панов

Катедра “Информатика”

Лекция 2 (10)

Функции и псевдоними в C++.

Да поговорим отново за локалните променливи

Да си припомним какво е локална променлива. (която е обявена вътре в една функция). Да припомним какво е блок. {}. Най-разпространения програмен блок е функцията. Блоковете обаче (но не и функциите) могат да бъдат вложени един в друг. **Ако една променлива е определена вътре в един блок, тя е локална по отношение на същия блок**. Т.е. сега доразвихме понятието „локална променлива“. С други думи, локалната променлива е неизвестна извън пределите на своя блок. Същото важи не само за променливите, но и за именованите константи и масивите, да ги наречем тук с общо име обекти. **Следователно, инструкции извън блока не могат да получат достъп към обект, дефиниран вътре в блока**.

Важно е да се разбере, че една локална променлива съществува само по време на изпълнение на програмния блок, в който тя е обявена. Това означава, че същата се създава при изпълнение на инструкцията, която я

дефинира **и се унищожава при излизане от блока**. А след като се унищожава, нейната стойност се губи! Прост пример, забележете променливата **x**:

```
#include <iostream> // nashata 51-a programa
using namespace std;//  lokalni promenlivi
int main() {
	int choise, x=11;
	cout <<"Vavadete malko cialo + chislo choise = " ;
	cin >> choise;
	if (choise> 0 && choise <6){
		int x=10, y=7, z;
		z= x+y; // polzva se vytreshnata x
		cout << "z=" << z << "\n"; // pozvoleno
	} cout << "x=" << x << "\n"; //  moje!
//	cout << "z=" << z << "\n"; // ne moje, shte dade greshka
	return 0;
}
```

Променлива може да се обяви и в раздела на инициализацията на цикъла **for** (доста често) или **управляващ-израз** на инструкции **if, switch** или **while** (твърде рядко). Променлива, обявена в една от тия инструкции, има област на видимост, равна на блока, управляван от същата инструкция. Например:

```
#include <iostream>
using namespace std;
int main()
{ // Promenlivata i e lokalna za cikyla  for.
      for (int i = 1; i<=10; i++ ) {
            cout << i << " ";
            cout << "na kub e rawno na " << i * i * i<< "\n";
      }
// i = 10; // ne moje, tuka i e neizvestna
      return 0;
}
```

И все пак в дадения пример дали може `i = 10;` често зависи и от компилатора!

Формални параметри и аргументи на функция

Ще дадем формално описание на функция като допълнение към вече казаното в трета лекция. Да не забравяме, че програмата винаги започва с извикването на функцията **main**.

Една функция се дефинира по следния начин:

```
тип_на_резултат име_на_функция (списък-формални-параметри)
{
/*
 Между тия две къдрави скоби се разполага кодът,
 който се изпълнява от функцията.
*/
}
```

тип_на_резултат - определя типа на стойността, връщана от функцията. Той може да бъде всеки един от типовете данни, с изключение на масив (т.е. цял масив). Ако една функция не връща стойност, се използва тип **void**.

Една функция връща стойност с помощта на инструкция **return**. Тя има следния синтаксис:

return връщана-стойност; (return; ако тип_на_резултат е **void**)

връщана-стойност е израз (константа или променлива като частен случай)

Типът на връщана-стойност трябва да е съвместим с дефинирания в заглавието на функцията тип_на_резултат. Под "съвместим" се има предвид същия тип или тип, който може да се преобразува към тип_на_резултат. Правилата за преобразуване са същите като при присвояване. **return** води до прекратяване на изпълнението на кода на функцията и програмата продължава с изпълнението на инструкцията след извикването на функцията. Една функция може да има повече от един return.

Името на функцията следва същите правила като при променливите. списък-параметри - определения на променливи, разделени със запетая. Т.е.

тип име-параметър1, тип име-параметър2, ... , тип име-параметърN

Ако функцията няма параметри списъкът е празен. Независимо от това, че тия променливи изпълняват специалната задача по получаването на стойностите на аргументите **ние можем да ги използваме във функцията подобно на локалните променливи**. Например, някъде вътре във функцията да им се присвои нова стойност. Областта на видимост на формалните параметри е ограничена в рамките на тяхната функция.

Извикването на функция става така:

име_на_функция (списък-аргументи); **(Заб. аргументи = фактически параметри)**

където списък-аргументи представлява стойности, разделени със запетаи, които се подават на функцията. Ако функцията няма формални параметри то тя не приема стойности, т.е. списък-аргументи е празен.

При извикването на функцията всеки от аргументите се копира в съответната променлива от списък-формални-параметри.

Обърнете внимание на следните термини: **аргумент** и **параметър**. Аргументът може да бъде константа, променлива или всеки допустим израз, който се подава на функцията при нейното извикване. Параметърът е променливата, която приема тоя аргумент. Ето защо аргументът и съответният параметър трябва да са от съвместими типове. (Броят на аргументите и параметрите трябва да е еднакъв). Под „съвместим“ се има предвид същият тип или тип, който може да се преобразува към типа на параметъра. Правилата за преобразуване са същите като при присвояване.

Частта на функцията, затворена между { и } се нарича тяло на функцията. Тялото на функцията съдържа инструкциите, изпълнявани от тази функция.

Всяка функция е добре да има прототип (вече разгледан в лекция 3), който да се намира преди първото извикване на функцията. Обикновено

прототипите се поставят в началото на файла. Функцията main е единствената функция, която не се нуждае от прототип, защото е определена от стандарта.

Една функция може да се извика като самостоятелна инструкция, т.е. да завършва с точка и запетая, но може да бъде извикана и в израз, като върнатата от нея стойност участва в изчислението на израза.

Механизъм на подаване на аргументи на функция

Аргументите могат да се подават (предават) на функциите по два начина:

- Предаване на аргументи по стойност. Стойността на аргумента се копира в параметъра. Всяка по-нататъшна промяна на параметъра в тялото на функцията не влияе на стойността на аргумента;

- Предаване на аргументи по адрес. При тоя начин на функцията се предава (копира) не стойността, а адреса на аргумента. А щом е адрес това

означава, че формалният параметър трябва да бъде указател и функцията може да изменя чрез него стойността на аргумента. Подаването на адрес на аргумент е начин една функция да върне повече от един резултат в извикващата я програма. Технически операторът return може да върне само една стойност.

Независимо как се подава аргументът действията, които се извършват, са аналогични на действията при присвояване. Ако типът на аргумента е различен от този на параметъра, компилаторът ще се опита да го преобразува. Ако преобразуването е невъзможно, ще се генерира грешка при компилиране.

Следващата програма_53 показва как изменението на параметъра в тялото на функцията не влияе по никакъв начин на аргумента. Стойността на аргумента **a** се копира в параметъра **x**. Всякакви по нататъшни действия върху **x** не засягат аргумента **a**.

```
#include <iostream> // nashata 53-a programa
using namespace std; // predavane na parametri po stoinost
void print_one_value(int x);
int main(void) {
	int a = 10;
	cout << "in main:a = " << a << "\n";
	print_one_value(a);
	cout << "in main:a = " << a << "\n";
	return 0;
}
void print_one_value(int x) {
	x = 20; // promianata na x ne vliiae na a
	cout << "in print_one_value  x = " << x << "\n";
}
```

В програма_54, подобна на предишната, указателят px променя стойността на аргумента **a**.

```
#include <iostream> // nashata 54-ta programa
using namespace std; // predavane na parametri po adres
void print_one_value(int *px);
int main(void) {
	int a = 10;
	cout << "in main:a = " << a << "\n";
	print_one_value(&a);
	cout << "in main:a = " << a << "\n";
	return 0;
}
void print_one_value(int *px) {
	cout << "begin in print_one_value:  *px = " << *px << "\n";
	*px = 20; // pametta sochena ot ukazatelia (argumenta a) se promenia
	cout << "  end in print_one_value:  *px = " << *px << "\n";
}
```

Във втория вариант на програмата изменението на стойността на указателя px не изменя стойността на указателя аргумент pa.

```
#include <iostream> // nashata 54b programa
using namespace std; // predavane na parametri po adres
void print_one_value(int *px); // sega argumentyt e ukaztel
int main(void) {
      int a = 10;
      int *pa; pa = &a; // ili napravo int *pa = &a;
      cout << "in main:a = " << a << "\n" << "\tpa=" << pa << "\n";
      print_one_value(pa);
      cout << "in main: a = " << a << "\n" << "\tpa=" << pa << "\n";
      return 0; }
void print_one_value(int *px) {
   cout<<"begin in print_one_value *px = "<<*px<<"\tpx="<<px<< "\n";
      *px = 20; // pametta sochena ot ukazatelia (argumenta a) se promenia
      cout << " in the middle of print_one_value *px = " << *px ;
      px = NULL; // adresyt v ukazatelia pa ne se promenia
      cout << "\nend in print_one_value px=" << px << "\n";
}
```

Следващият пример_55 показва как да защитим обекта, сочен от указателя-параметър, от изменение в тялото на функцията.

```
#include <iostream> // nashata 55-ta programa
using namespace std; // predavane na parametri po adres
void print_one_value(const int *px); // sega argumentyt e const ukaztel
int main(void) {
	int a = 10;
	int* pa = &a;
	cout << "in main: a = " << a << "\n" ;
	print_one_value(&a);
	return 0;
}
void print_one_value(const int *px) {
	cout << "in print_one_value: *px = " << *px << "\n";
//	*px = 20; // Greshka!
}
```

В случаи, когато обект се подава на функция по адрес и тая функция не трябва да го изменя е препоръчително параметърът да се декларира като указател към константна-променлива. Тогава всеки случаен опит да се промени обектът от функцията ще бъде засечен от компилатора при компилиране на кода. За да се убедите в това, махнете коментара от инструкцията *px = 20; и компилирайте. Ще получите съобщение за грешка.

Масиви като параметри на функции

Масивите също могат да се предават като аргументи на функция. Името на масива служи като аргумент. Вече знаем, че името на масив е указател. Затова и параметърът, който го приема е указател към същия тип данни. **Това води до следния важен извод: когато масив се предава на функция, това, което в действителност се предава, е адресът на първия елемент на масива, а не целият масив.**

Езикът C++ допуска три алтернативни форми на обявяване на функция, която приема масив за аргумент:

1) тип име_на_функция(тип име_на_масив[размер]);

2) тип име_на_функция(тип име_на_масив[]);

3) тип име_на_функция(тип * име_на_указател);

В първата форма, въпреки че е посочен и размера на масива, той се игнорира от компилатора. Ето защо размерът може да се изпусне както е показано във втората форма. В крайна сметка първите два варианта се трансформират до третата форма. Тя най-точно отразява механизма на предаване на масив на функция.

След като функцията има адреса на масива, тя може да го променя (масива). Пример за това е следващата програма_56.

```
#include <iostream> // nashata 56-ta programa
using namespace std; // predavane (imeto) na masiv kato argument na funkcia
const int SIZE = 5;
void init_array(int p[], int initializator); void print_array(int p[]);
int main() {
	int arr[SIZE];
	init_array(arr, 50); print_array(arr);
	init_array(arr, 72); cout<<"\n"; print_array(arr);
	return 0;
}
void init_array(int p[], int initializator){
	for (int ind = 0 ; ind < SIZE ; ind++){  p[ind] = initializator+ind;}
}// wsichki elem na masiva se inicializirat
void print_array(int p[]){// syshtoto kato void print_array(int *p)
	for (int ind = 0 ; ind < SIZE ; ind++){
		cout << "["<<ind<<"]="<<p[ind]<<"\n";
	}
}
```

Също както една функция не може да приеме цял масив като аргумент, така функцията не може да върне цял масив като резултат. Тъй като име на масив е указател, то **конструкция от вида return a; където a е име на масив, в действителност връща адреса на първия елемент на масива**. Затова и типът на връщания резултат в дефиницията на функцията трябва да бъде указател.

Примерът, (програма_57) който следва е по-общ, (да не забравяме, че името на масив е константен указател) ще разгледаме как **return** връща указател. Разбира се, **тип_на_резултата** пред името на функцията също трябва да бъде указател. Функцията **find_substr** търси дали даден низ се среща в друг по-голям. Ако поднизът е намерен се връща указател към неговото начало в големия низ, иначе се връща **NULL**. По същия начин работи и готовата функция **strstr.** Може да се каже, че тук разглеждаме една възможна имплементация на **strstr**.

```
#include <iostream> // nashata 57-ma programa
#include <cstring> // zaradi strstr()
using namespace std; // primer kak funkcia wryshta ukazatel
char *find_substr(char *str, char *sub);
int main(){
	char *substr;
	substr = find_substr("Bodro da varvim napred", "da");
	cout << "nameren podniz s nachata programa: " <<  substr;
	substr = strstr("Bodro da varvim napred", "da");
	cout << "\nnameren podniz s gotovata  strstr: " <<  substr;
	return 0;
}
char *find_substr(char *str, char *sub){ // funkciata vryshta ukazael kym
	int t; // namerenia podniz ili NULL, ako takyv ne e nameren
	char *p, *p2, *start;
	for(t=0; str[t]; t++) {
		p = &str[t]; // sochi tekushtia simvol na golemia niz
		start = p; // napravete variant na find_substr bez start
		p2 = sub; // otnovo otivame v nachaloto na podniza i tyrsim pak
		while(*p2 && *p2++ == *p++) { // proverka za syvpadenie
```

```
                if ( !*p2){
// Ako e dostignat kraia na p2-podniza znzchi sme namerili podniza
                  return start; // Vryshtame nachaloto na namerenia podniz v niza
                }
            }
        }
        return NULL; // podnizyt ne e nameren
}
```

Масивът, аргумент на функция, може да е двумерен или с по-голяма размерност. Такъв пример е следващата програма_58a. **Всяко измерение освен първото на масив, който е формален параметър трябва да бъде явно указано**. И за двумерни масиви формалният параметър може да е указател, как става ще бъде показано на упражнения (програма_58b).

```
#include <iostream> // nashata 58a programa
#include <cmath> //sybirane na ednotipni matrici
using namespace std; // tova e programa 34, no sega s funkcii
const int M = 2; // broi na redovete na matricata
const int N = 3; // broi na kolonite na matricata
void input_array(float p[][N]); void print_array(float p[][N]);
void sum_arrays(float p1[][N],float p2[][N],float p3[][N]);
int main() {
	float a[M][N], b[M][N], c[M][N];// c e rezultatnata matrica
	input_array(a);
	input_array(b);
	sum_arrays(a, b, c);
	cout << "Rezultatnata matrica sled sybiraneto: \n";
	print_array(c);	return 0;
}// samo pyrvata razmernost na masiva moje da ne se zadawa !!
void input_array(float p[][N]){
 cout << "Vavedete stoinosti za matrica s razmeri "<<M<<" na "<<N<< "\n\n";
	for (int i=0; i < M;i++){ // vavejdane na dvumeren masiv
		for (int k=0; k < N;k++){
			cout << "["<<i<<"]["<<k<<"]=";
```

```
                cin >> p[i][k];
            }
        } cout << "\n";// za preglednost
}
void sum_arrays(float p1[][N],float p2[][N],float p3[][N]){
        for (int i=0; i < M;i++){ // sybirane na matricite
            for (int k=0; k < N;k++){
                p3[i][k] = p1[i][k] + p2[i][k];
            }
        }
}
void print_array(float p[][N]){
        for (int i=0; i < M;i++){ // izvejdane na dvumeren masiv
            for (int k=0; k < N;k++){  cout <<p[i][k]<< "\t";
            }//"\t"  za pravilna podredba
            cout << "\n"; // za poredovo otpechatvane
        }
}
```

Клас памет на функциите

Дотук всяка от нашите програми се съдържа в един единствен файл. На практика при големи програми както броят на сорс файловете, така и тия на заглавните файлове може да бъде значителен. В такива случаи се използват две ключови думи - **static** и **extern**.

Когато една функция ще се извиква само във файла, в който е определена, тя може да се направи "невидима" за всички останали файлове. За целта пред прототипа и определението ѝ трябва да се постави ключовата дума **static**.

static тип_на_резултата име_на_функция(списък-формални-параметри);

Ползите са две: Първо, можем да имаме функции с едно и също име и параметри (но с различно тяло) в отделните файлове. Втори, при опит за

извикване на функцията от друг файл на програмата компилаторът ще сигнализира за грешка.

Ако една функция ще се извиква от различни файлове, то всеки файл трябва да включва и прототипа на функцията (наричан често и декларация) преди точката, в която функцията се извиква. Ключовата дума extern не е задължителна, т.е. следните две форми на прототипи са еквивалентни.

тип_на_резултата име_на_функция(списък-формални-параметри);

extern тип_на_резултата име_на_функция(списък-формални-параметри);

Някои от програмистите ползват и двете форми, като избират втората когато функцията наистина е определена в друг файл, а първата форма когато функцията е определена в текущия файл.

Псевдоними (Алтернативни имена)

Алтернативното име или псевдонима представлява начин да зададем още едно име на дадена променлива. Псевдонимът трябва задължително да се инициализира при обявяването му – т.е. на него се присвоява адреса на някоя вече обявена променлива. **След това псевдонимът може да се употребява навсякъде, където е позволено да се ползва променливата, на която е взет адреса.** На практика няма никаква разлика дали ще се използва името на променливата или нейния псевдоним. Следва проста програма (59-а) за онагледяване как се създава и ползва алтернативно име. Обърнете внимание, че адресите на променливата и нейния псевдоним са еднакви.

Ползата от такива псевдоними е минимална, даже в определени моменти може да доведе до объркване. Реалната употреба на псевдонимите е като формални параметри на функция. Основно правило: **Псевдонимът параметър автоматично получава адреса на съответстващия аргумент.**

```
using namespace std; // nashata 59-ta programa
#include <iostream>// primer za psevdonim
int main()
{
      int j, k;
      int &i = j; // syzdavame psevdonimyt i
      j = 10;
      cout << j << " " << i;
      k = 200;
      i = k; // stoinostta na prom. k se kopira v prom. j
      cout << "\n" << j; // izvejda se  200
      cout<<"\n&i=" <<&i <<"\t&j=" << &j << endl;// adresite sa ednakvi      return
0;
}
```

Т.е. не е нужно ние явно да указваме адреса с оператора & при извикване на функцията. При изпълнение на кода на функцията, а именно при

изпълнение на операции над параметър-псевдоним, се прави автоматично **„дереференциране“**, поради което на програмиста не се налага да използва операторите, работещи с указатели. На английски **параметър-псевдоним** е **reference parameter** и **дереференциране означава когато се среща псевдонима да се работи със стойността, която се намира на съответния адрес**. Самото обявяване на един формален параметър като псевдоним става с &, т.е. по същия начин, както при обикновения псевдоним, вече разгледан в програма_59. **тип &псевдоним**

Ще отбележим, че псевдонимите не се поддържат в C, а само в C++.

Следва проста програма_60 с използването на параметър-псевдоним. Освен при обявяването си, псевдонимът никъде не изисква повече употреба на **&**. Т.е. той е по-удобен от указателя, където трябва да се внимава кога се употребява **&** и кога ***.

```
using namespace std; // nashata 60-a programa
#include <iostream> // izpozlvane na parametyr psevdonim
void f(int &i);
int main(){
	int a = 19;
	cout << "old value a=" << a << '\n';
	f(a);
	cout << "new value a=" << a << '\n';
	return 0;
}
void f(int &i){ // // ! zabelejete &
	i = 47; // promeniame STOINSOTTA na argumenta
}// zadaden pri izvikvane na funkciata
```

Важно: Функция може да връща и псевдоним с return. Това означава, че тя връща неявен указател към стойността, предавана от нея в инструкция **return**. Това открива нови възможности, показани в програма61.

```
using namespace std; // nashata 61-va programa
#include <iostream> // vryshtane na psevdonim  vyv funkcia
double &f1(); // funkciata vryshta psevdonim
double val = 100.7; // globalna promenliva
int main(){
      double newval;
      cout << f1() << '\n'; // izvejda se stoinostta na val.
      newval = f1(); // na newval se prisvoiava stoinostta na val
      cout << newval << '\n';
      f1() = 77.1; // !!! Promeniame stoinostta na val
      cout << f1() << '\n'; // izvejda se novata stoinost na val.
      return 0;
}
double &f1 () {
      return val; // vryshta se neiaven ukazatel kym val.
}
```

За да разберем как работи програмата е достатъчно да осмислим коментара във функцията **f1()**. Ако приемем, че неявният **глобален** указател има име **pval** то той има вида double ***pval** = & **val;** Инструкция return val; връща **pval** (т.е. адреса на **val**). Този адрес се използва за косвен достъп до променливата **val** в зависимост от текущата инструкция. Можем да мислим, че навсякъде където се извиква **f1()** резултатът е ***pval**. Т.е. можем да считаме, че f1() = 77.1; всъщност е *pval = 77.1. Аналогично cout << f1() става cout << *pval както и newval = f1(); всъщност е newval = *pval;

Изводът е, че псевдоним и указател имат много общо. Следва изброяване на **ограниченията при използване на псевдоними:**

- Не можем да създаваме псевдоним на псевдоним. (указ. към указ. може)
- Не можем да създаваме масив от псевдоними. (масив от указатели може)
- Не можем да създаваме указател към псевдоним. (за **полета** в др. лекция)
- Не можем да използваме псевдоним за битови полета на структура.