



ФУНКЦИИ

Изготвена от Никола Светославов

Summary



Какво е функция и защо да я използваме.



Референция и указатели. Референция във функции



Overloading на функции и двусмислици.



Параметри по подразбиране



Function Declaration vs. Definition

Какво е функция ?

- (именувано)парче код, което изпълнява някакво действие.
- Може да бъде извикано от друг код, да бъде използван многократно.
- Може да приема параметри и да връща даден резултат/стойност.
- Освобождава място в `main()`
- `main()` е функция.

Синтаксис

- <сигнатура> <идентификатор> ([<формални_параметри>]) { <тяло> }
- <сигнатура> - [<тип_результат> | void]
 - void = празен тип, не връща резултат
 - Ако типът на резултата се пропусне, подразбира се int
- Ако функция със сигнатура различна от void не връща стойност се получава грешка при компилация.
- Ако има формални параметри, то трябва да се специфицира типът им.
- Ако параметър е примитивен тип данна или някакъв обект, то се създава нов обект в scope-а на функцията!

Извикване на ф-я

- <име>(<фактически_параметри>);
- Извикването на функция всъщност е операция с много висок приоритет
- Използването на функции/Извикването на функции е почти същото като използването на променливи.
- Пишем **()** след функцията, като може да съдържа параметри.

Оператор return

- Оператор за връщане на резултат на функция
- Типът на <израз> се съпоставя с типа на резултата на функцията ако се налага, прави се преобразуване на типовете
- Работата на функцията се прекратява незабавно
- При сигнатура void, return не връща нищо, а просто прекъсва функцията (не е задължителен)

Пример

```
#include<iostream>

void helloWorld() {

    std::cout << "Hello World!" << std::endl;
}
```

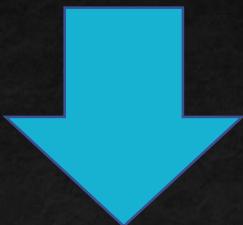
```
int main() {

    helloWorld();

    return 0;
}
```

Пример - Функция намираща сбора на 5 числа

```
int Sum (int a, int b, int c, int d, int e) {  
    int temp = a + b + c + d + e;  
    return temp;  
}
```



```
int Sum (int a, int b, int c, int d, int e) {  
    return (a + b + c + d + e);  
}
```

Пример

```
bool ValidateData(int a) {  
    if (a >= 1000) {  
        return true;  
    }  
    if (a % 2 != 0) {  
        return false;  
    }  
  
    // Грешка при компилиране(undefined  
    behaviour), защото не всички възможни  
    изходи връщат стойност
```

Q&A

- Q: Примерите дотук изглеждат тривиални и прекалено лесни, за да се наложи да използваме функция. Какво ще стане ако просто си ги въвеждам всеки път?
- A1: Кодът ти ще е претрупан с повтарящи се фрагменти, а това намалява качеството на кода.
- A2: Ако решиш да промениш нещо ще тряба да пренаписваш кода навсякъде. Това е загуба на време, а и може да е източник на грешки.

Пояснение относно параметрите

- Не са задължителни
- Създават се нови обекти, като при примитивните типове данни това е много бърза операция, но при някои други данни може да е много бавно.
- **Новите обекти са равни на оригиналните, но не са свързани с тях.**
- Ако промените временен обект във функцията, оригиналният не се променя.
- Подредбата им е от значение.
- В тялото на функцията може да се създават нови променливи с имена на параметри, но те **НИКОГА** няма да бъдат достъпни.

Пример

```
void swap(double a, double b) {  
    double c = a;  
    a = b;  
    b = c;  
}
```

//Нищо няма да се случи, защото a и b са нови временни обекти.

//Тези променливи НЕ СА свързани с променливите, които сме подали като параметри!!!

Пример

```
int main() {  
    int a = 5;  
    example(a);  
    return 0;  
}
```

0x34 - a									
0000	0000	0000	0101						

Пример

```
void example(int numb) {  
    numb+=5;  
}
```

0x34 - a													
0000	0000	0000	0101										
				0xA21 - numb									
						0000	0000	0000	0101				

Пример

```
void example(int numb) {  
    numb+=5;  
}
```

0x34 - a										
0000	0000	0000	0101							
				0xA21 - numb						
						0000	0000	0000	1010	

Пример

```
int main() {  
    int a = 5;  
    example(a);  
    return 0;  
}
```

0x34 - a									
0000	0000	0000	0101						

Reference/Pointer



Reference/Референция

Алтернативно име за съществуваща променлива. Променлива може да бъде декларирана като референция чрез '&'.

Ако функция получи референция към променлива, тя може да променя(modify) стойността на променливата(директно).

Може да предотврати копирането на големи структури от данни.



Pointer/Указател

Променлива, която пази адрес(memory address) като стойност.

Променливата на указател сочи към типа данни, от същия тип, която е и тя, и се създава чрез оператор '*'. Адресът на променливата, с която работите, се присвоява на указателя.

Пример

```
int number = 5;  
int* ptr = &number;// A pointer  
variable, with the name ptr, that  
stores the address of number  
  
// Output the value of number  
std::cout << number << "\n";  
  
// Output the memory address of  
number  
std::cout << &number << "\n";  
  
// Output the memory address of  
number with the pointer  
std::cout << ptr << "\n";
```

Пример

```
void swap(double &a, double &b) {  
    double c = a;  
    a = b;  
    b = c;  
}  
  
//Тези променливи са свързани с  
променливите, които сме подали като  
параметри!!!
```



First Skill

Second Skill

Overloading

Q: Възможно ли е да имам функция, която да прави повече от 1 действие в зависимост от подадените параметри

■ Пример:

Sum(1,2,3)

Sum(1,2,3,4)

Sum(1.55,1.2)

A: Да, това се нарича function overloading



Third Skill

Conclusion

PAGE 20



First Skill

Second Skill

Overloading

- Една функция може да има безброй много overloads
- При извикване на функцията, компилаторът се грижи да намери правилният overload на функцията
- Компилаторът може да направи преобразуване на данните ако се налага



Third Skill

Conclusion



First Skill

Second Skill

Overloading

- Ако не намери подходящ се получава грешка при компилиране
- Ако намери повече от 1 подходящ се получава грешка за двусмислие



Third Skill

Conclusion

PAGE 22

Пример

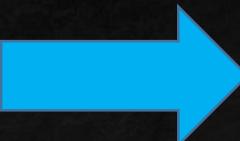
```
void cout(char a){std::cout<<a;} //1
void cout(int a){std::cout<<a;} //2
void cout(char a, int b){std::cout<<a<<'-'<<b;} //3
void cout(double a, char b){std::cout<<b<<'-'<<a;} //4
void cout(bool a){std::cout<<a;} //5
void cout(char a, bool b, int c){std::cout<<a<<b<<c;} //6
void cout(const int a){std::cout<<a;} //7
void cout(char a, unsigned b){std::cout<<a<<'-'<<b;} //8
char cout(char a){return a;} //9
```

Пример

```
void cout(char a){std::cout<<a;} //двусмислие с 9  
  
void cout(int a){std::cout<<a;} // двусмислие със 7  
  
void cout(char a, int b){std::cout<<a<<'-'<<b;}  
  
//двусмислие с 8  
  
/*-----*/  
  
void cout(double a, char b){std::cout<<b<<'-'<<a;}  
  
void cout(bool a){std::cout<<a;}  
  
void cout(char a, bool b, int c){std::cout<<a<<b<<c;}  
  
/*-----*/  
  
void cout(const int a){std::cout<<a;} // двусмислие с 2  
  
void cout(char a, unsigned b){std::cout<<a<<'-'<<b;}  
  
// двусмислие с 3  
  
char cout(char a){return a;} // двусмислие с 1
```

Как да ги променим?

```
void cout(char a, int b)  
{ std::cout<<a<< " - " <<b; }  
  
void cout(char a, unsigned b)  
{ std::cout<<b<< " - " <<a; }
```



```
void cout(char a, int b)  
{std::cout<<a<< " - " <<b;}  
  
void cout(unsigned b, char a)  
{std::cout<<b<< " - " <<a;}  
  
//разменяме unsigned int и char parameters
```

- Важно! За компилатора има значение подредбата на параметрите. Ако спрямо дадената подредба няма отговаряща функция се получава грешка при компилация
- Другите 2 двусмислия няма как да се отстроят така, а чрез смяна на името на функцията. //или добавяне на параметър, който не се ползва(**ЛОША ПРАКТИКА, НЕ го правете**)
- Пример: `char cout(char a, bool useless){return a;}` **//лоша практика**

Параметри по подразбиране

- Възможно е да имате програма, в която 90% от случаите подават един и същ параметър на дадено място
- С++ позволява да имате стойност по подразбиране за 1 или повече параметри, които не се налага да уточнявате при извикване на функцията
- Синтаксис:

```
void Cout(int a, int b = 5){std::cout<<a << " " << b;}
```

```
Cout(4); // 4 5
```

```
Cout(3, 6); // 3 6
```

- Параметрите по подразбиране трябва винаги да са в края!!!

Параметри по подразбиране

```
void Cout(int a, int b = 5, char c = 't')  
{  
    std::cout << a << " " << b << " " << c;  
}
```

```
Cout(4); // 4 5 t
```

```
Cout(3,6); // 3 6 t
```

```
Cout(3, '0'); // 3 48 t
```

- '0' има стойност 48 в ASCII => компилаторът го разглежда като int със стойност 48
- Параметрите по подразбиране винаги са в последователността, в която са дефинирани, не могат да се прескачат

Function Declaration vs. Definition

**Declaration –
function's name,
return type and
parameters**

Declaration – казва на компилатора,
че така функция съществува.

Definition – казва на компилатора, какво всъщност прави
тази функция(нейната функционалност)

Една функция може да бъде декларирана, но да не бъде
дефинирана. Получаваме компилационна грешка, ако тази
функция бъде извикана.