

Функции

Какво е функция ?

Нека разгледаме следната програма, която намира простите числа от 1 до `n`:

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int n = 20;

    for(int i = 1; i < n; i++)
    {
        bool is_prime = true;
        for(int j = 2; j <= sqrt(i); j++)
        {
            if(i % j == 0)
            {
                is_prime = false;
                break;
            }
        }
        if(is_prime)
        {
            cout << i << " ";
        }
    }
    cout << endl;
    return 0;
}
```

Нека разгледаме частта, която проверява дали дадено число е просто или не:

```
bool is_prime = true;
for(int j = 2; j <= sqrt(i); j++)
{
    if(i % j == 0)
    {
        is_prime = false;
        break;
    }
}
if(is_prime)
{
    cout << i << " ";
}
```

Сама по себе си, тази част от кода може да бъде представена като отделна малка програма, която при подадено число, връща отговор дали това число е просто. За нас е важно да знаем две неща - какво приема тази "програма" и какво връща.

Тези "малки програми" в програмирането могат да бъдат обособени като отделни едници, с помощта на т.нар **функции**

Функцията е блок от код, който върши дадена работа, може да приема и връща стойности. Примери за функции за `main()`, `abs()`, `sqrt()`

Синтаксис

Синтаксиса на функцията изглежда по следният начин:

```
тип_на_функцията име_на_функцията(аргументи)
{
    //тяло на функцията
    return резултат;
}
```

Където:

- тип_на_функцията е типа на резултата (пример: `int`, `double`, `char`, `void` и др.)
- име_на_функцията е името, с което ще извикваме функцията
- аргументи - 0 или повече аргумента - т.е. какво приема функцията като данни
- резултат - какво връща функцията (**бележка:** функцията може да не връща резултат)

Примерът с намирането на това дали едно число е просто, може да бъде представен чрез функция по следният начин:

- Типът на функцията, т.е. резултата ѝ трябва да ни казва дали дадено число е просто или - подходящия тип е `bool`
- Името на функцията може да бъде `is_prime`
- Аргумента тук е един - числото, за което ще проверяваме дали е просто
- Резултата - ще връщаме `true` ако числото е просто или `false` ако не е. Това връщане се случва с оператора `return`

Функциите се декларират извън `main()` - т.е. не може да имаме функция вътре в тялото на друга функция (`main()` също е функция)

```
#include <iostream>
#include <cmath>
using namespace std;

bool is_prime(int to_check)
{
    bool is_prime = true;
    for(int j = 2; j <= sqrt(to_check); j++)
    {
        if(to_check % j == 0)
        {
            is_prime = false;
            break;
        }
    }
    return is_prime;
}
```

```

}
int main()
{
    int n = 20;

    for(int i = 1; i < n; i++)
    {
        if(is_prime(i) == true)
        {
            cout << i << " ";
        }
    }
    cout << endl;
    return 0;
}

```

Вече като искаме да проверим дали дадено число е просто, извикваме функцията `is_prime()`.

Оператор return

Оператора `return` служи, когато искаме да приключим изпълнението на дадена функцията и да върнем даден резултат (според типа на функцията, има и функции, които не връщат резултат)

```

bool is_even(int number)
{
    if(number % 2 == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Важно е да отбележим, че когато имаме условен оператор или оператор за цикъл, **всички** разклонения на програмата трябва да имат оператор `return`.

Още примери за функции:

```

int sum(int first, int second)
{
    int result = first + second;
    return result;
}

```

```
bool is_triangle(int first, int second, int third)
{
    bool first_check = ((first + second) > third);
    bool second_check = ((second + third) > first);
    bool third_check = ((first + third) > second);

    return (first_check && second_check && third_check);
}
```

В този пример, проверяваме дали три числа са страни на валиден триъгълник

Проверяваме всяка от комбинациите поотделно, а после връщаме стойността на израза `(first_check && second_check && third_check)`, т.е. дали и трите променливи имат стойност `true` (ако една има стойност `false`, то стойността на целият израз ще е `false`)

```
double area_rectangle(double a, double b)
{
    return a * b;
}
```

Void функции

По-горе беше споменато, че има функции, които не връщат нищо. Типът на тези функции е `void`.

При `void` функциите, оператора `return` не е необходим.

Примери за `void` функции:

```
#include <iostream>

using namespace std;

enum days
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};

void print_day(days current_date)
{
    switch (current_date)
    {
        case Monday:
            cout << "Monday";
            break;
        case Tuesday:
            cout << "Monday";
            break;
        case Wednesday:
```

```

        cout << "Monday";
        break;
    case Thursday:
        cout << "Monday";
        break;
    case Friday:
        cout << "Monday";
        break;
    case Saturday:
        cout << "Monday";
        break;
    case Sunday:
        cout << "Monday";
        break;
    default:
        cout << "Not a day";
        break;
    }
}
int main()
{
    return 0;
}

```

Дефиниция vs Декларация

Може да извикваме функции, само когато са декларирани над функцията от която ги извикваме

```

void foo()
{
    //Do something
}

int main()
{
    foo();
    return 0;
}

```

, но не и:

```

int main()
{
    foo();
    return 0;
}

void foo()
{
    //Do something
}

```

За да се справим с този проблем, може да декларираме сигнатурата на функцията в началото на кода, а по-долу да напишем нейното тяло.

Пример:

```
void foo();

int main()
{
    foo();
    return 0;
}

void foo()
{
    //Do something
}
```

Област на видимост

Какво би станало ако сме декларирали променлива в някоя функция, и искаме да я използваме в `main()`, без да сме я върнали.

```
#include <iostream>

using namespace std;

void change_number()
{
    a = 15;
    int b = 200;
}

int main()
{
    int a = 1;
    f()
    b = 2;
    return 0;
}
```

Това ще даде грешка. Една променлива е "видима" само за блока в който е дефинирана (това включва и блоковете вътре в текущия блок)

Ако искаме да подадем променлива на функция, която да промени променливата, това става последният начин:

```
#include <iostream>

using namespace std;

int change_number(int to_change)
{
    to_change = to_change * 2;

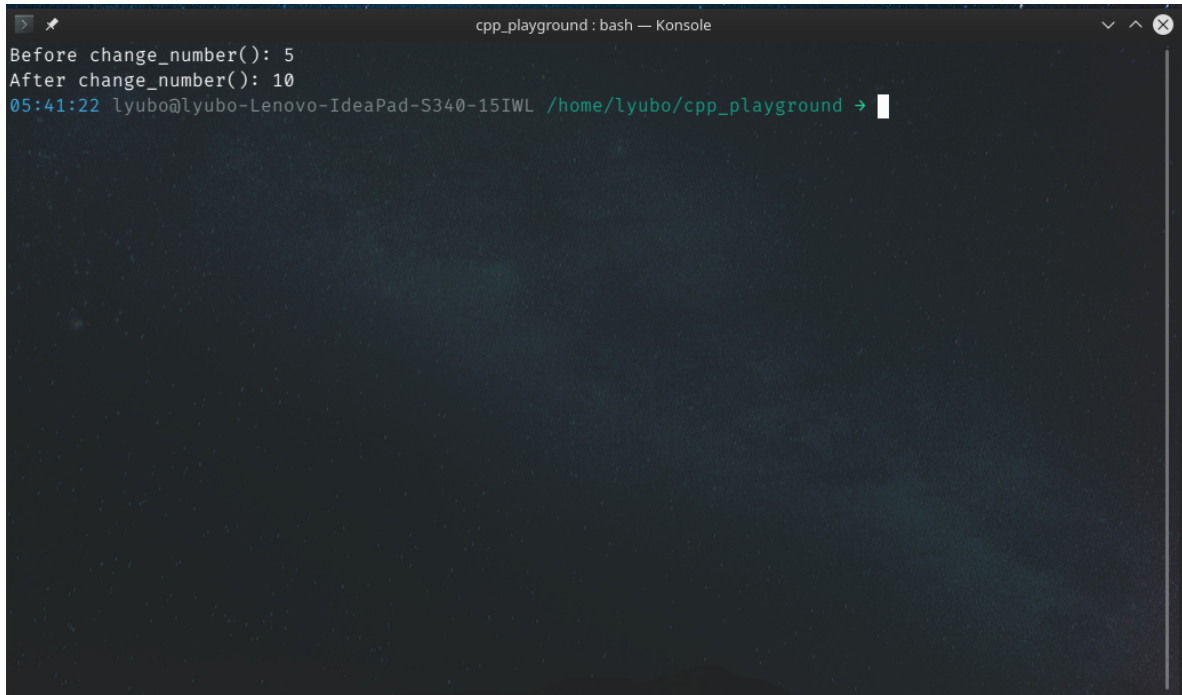
    return to_change;
}
```

```

int main()
{
    int variable = 5;
    cout << "Before change_number(): " << variable << endl;
    variable = change_number(variable);
    cout << "After change_number(): " << variable << endl;
    return 0;
}

```

Декларираме променлива от тип `int`, даваме ѝ стойност 5. След това я подаваме на функцията `change_number`. Функцията умножава променливата по две и я връща.



```

cpp_playground : bash — Konsole
Before change_number(): 5
After change_number(): 10
05:41:22 lyubo@lyubo-Lenovo-S340-15IWL /home/lyubo/cpp_playground →

```

Възможно е да декларираме променлива извън всички функции. Такива видове променливи се наричат глобални и са видими от всички функции. Не е добра практика да се използват.

```

#include <iostream>

using namespace std;

int global_variable = 5;

void change_global_variable()
{
    global_variable = 10;
}

int main()
{
    cout << "Global variable is currently: " << global_variable << endl;
    change_global_variable();
    cout << "Global variable is now: " << global_variable << endl;
    return 0;
}

```

```
cpp_playground : bash — Konsole
Global variable is currently: 5
Global variable is now: 10
05:41:55 lyubo@lyubo-Lenovo-IdeaPad-S340-15IWL /home/lyubo/cpp_playground →
```

Overloading

Нека разгледаме примера за проверка дали три страни могат да образуват триъгълник

```
bool is_triangle(int first, int second, int third)
{
    bool first_check = ((first + second) > third);
    bool second_check = ((second + third) > first);
    bool third_check = ((first + third) > second);

    return (first_check && second_check && third_check);
}
```

Забелязваме, че тази функция работи с аргументи от тип `int`. Какво става, ако искаме да направим още една функция със същото име, която прави същото нещо, само че работи с аргументи от тип `double`?

Това е позволено, с помощта на т.нар. "предефиниране на функции" (overloading).

Предефинирането на функции, позволява да съществуват функции с едно и също име, от един и същ тип, но с различни по брой и вид аргументи.

```
#include <iostream>

using namespace std;

bool is_triangle(int first, int second, int third)
{
    bool first_check = ((first + second) > third);
    bool second_check = ((second + third) > first);
    bool third_check = ((first + third) > second);

    return (first_check && second_check && third_check);
}

bool is_triangle(double first, double second, double third)
```



```

{
    bool first_check = ((first + second) > third);
    bool second_check = ((second + third) > first);
    bool third_check = ((first + third) > second);

    return (first_check && second_check && third_check);
}

int main()
{
    int side_a = 3, side_b = 4, side_c = 5;
    double side_d = 4.5, side_e = 6, side_f = 7.5;

    cout << is_triangle(side_a, side_b, side_c) << endl;
    cout << is_triangle(side_d, side_e, side_f) << endl;
    return 0;
}

```

```

double area_rectangle(double a, double b)
{
    return a * b;
}

double area_rectangle(double a)
{
    return a * a;
}

```

Параметри по подразбиране

Нека разгледаме следната функция

```

int sum(int a, int b)
{
    int result = a + b;
    return result;
}

```

C++ ни позволява да задаваме стойности по подразбиране на аргументите на функции. Това става по следният начин

```

#include <iostream>

using namespace std;

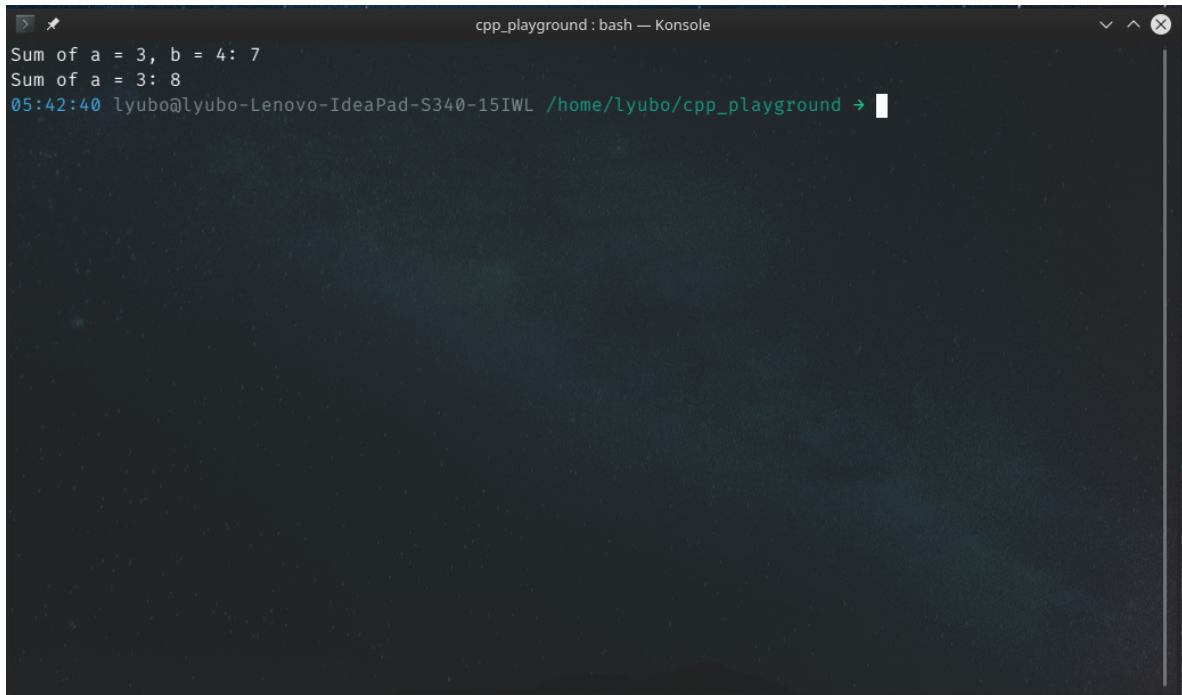
int sum(int a, int b = 5)
{
    int result = a + b;

    return result;
}

int main()
{
    cout << "Sum of a = 3, b = 4: " << sum(3, 4) << endl;
}

```

```
cout << "Sum of a = 3: " << sum(3) << endl;  
return 0;  
}
```



```
cpp_playground: bash — Konsole  
Sum of a = 3, b = 4: 7  
Sum of a = 3: 8  
05:42:40 lyubo@lyubo-Lenovo-IdeaPad-S340-15IWL /home/lyubo/cpp_playground →
```

Първия ред подаваме две числа, 3 и 4, и получаваме 7 като отговор. Когато подадем само едно число на функцията, се използва стойността на `b` по подразбиране - 5, и затова отговора на функцията е 8.

Важно е да отбележим, че параметрите по подразбиране се записват винаги като последни аргументи на функцията.

Inline

Когато извикаме някоя функция, в паметта се отделя място за нея, за нейните аргументи и за нейния код. Това понякога е бавна операция.

C++ ни предлага опцията вместо програмата да "отива" до функцията (при съответната и памет, т.н.), тялото на функцията да бъде копирано и поставено при всяко нейно извикване в кода - това става чрез ключовата дума `inline`

```
inline void foo()  
{  
    //код  
}
```

Когато функцията е дълга, или се използва повече пъти, това копиране на тялото би довело до спад на производителността на програма, вместо повишение. Най-често се използва за кратки функции. Модерните компилатори могат да игнорират тази декларация.

Задачи:

1. Да се пренапише задачата за калкулатор, като се използват функции
2. Да се напишат функции, която пресмятат лицата на следните фигури: квадрат, правоъгълник, триъгълник, трапец и кръг.

3. Да се напише функция, която намира най-голям общ делител на две числа
4. Да се напише функция, която решава кубично уравнение
5. Да се напише функция, която казва дали дадена точка (x, y) е в правоъгълник, зададен по две точки (x_1, y_1) и (x_2, y_2)