

||

26 февраль 2020

Базовые блоки программы на C++:

- 1 **Операторы** (operators) - символьные конструкции, обозначающие вызов действий и возвращающие некоторое значение в качестве результата.
- 2 **Выражения** (expressions) - последовательность операторов и их операндов, задающая определённые вычисления (возвращающие конкретное значение).
- 3 **Инструкции** (statements) - фрагменты программы, выполняющиеся в последовательном порядке.

Терминологическая путаница

Среди русскоязычной литературы вместо *операторов* может быть использован термин *операция*, а вместо *инструкций* — *операторы*.

Операторы

```
1 + - * /
2
3 & | ^
4
5 && ||
6
7 sizeof
8
9 ?:
10
11 (...)
```

Выражения

```
1 rate = 43
2
3 rate
4
5 2.0 + rate / 4.5
6
7 printf("My call is expression too")
```

Выражения и инструкции

Инструкции

```
1 int i1; // Инструкция—объявление
2
3 i1 = 10; // Инструкция—выражение
4
5 printf("i1 = %d\n", i1); // Инструкция—выражение
6
7 if (i1 > 50) { // Составная инструкция
8     // ...
9 } else {
10    // ...
11 }
12
13 int i2; // Инструкция—объявление
14 do { // Составная инструкция
15     i1++;
16     i2 = i1 * i2;
17 } while (i1 < 100);
```

Функция в C++ - проименованный набор инструкций, который может принимать *произвольное количество* значений, возвращать строго *одно* значение и быть вызван произвольное число раз другим фрагментом программы. Синтаксис определения функции:

```
[...] <тип_возвращаемого_значения>
      <имя_функции> ( <список_параметров> )
{
    [инструкции; ]
    return <значение>;
}
```

, где **список_параметров** - это набор пар *тип данных* и *идентификатор* каждого параметра, перечисляемых через запятую. Трехточия - специальные указания компилятору, по необходимости будут рассмотрены далее по курсу.

$$n! = 1 * 2 * 3 * 4... * n$$

Функции в C++. Пример

$$n! = 1 * 2 * 3 * 4... * n$$

```
1 double factorial(unsigned n)
2 {
3     double fact = 1.0;
4
5     for (unsigned i = 2; i <= n; ++i) {
6         fact *= i;
7     }
8
9     return fact;
10 }
11
12 ...
13 double fact8 = factorial(8);
14 printf("Factorial of 8: %f\n", fact8);
```


Функции в C++. Множественный **return**

Число операторов возврата **return** внутри функции - неограничено.

```
1 double factorial(unsigned n)
2 {
3     if ( n < 2) {
4         return 1.0;
5     }
6
7     double fact = 1.0;
8     for (unsigned i = 2; i <= n; ++i) {
9         fact *= i;
10    }
11
12    return fact;
13 }
```

Функции в C++. Пример

Возведение в степень:

number^n , n — целое

Функции в C++. Пример

Возведение в степень:

number^n , n — целое

```
1 double nth_degree(double num, int n)
2 {
3     double result = 1.0;
4     // Берём модуль от числа n
5     unsigned degree = abs(n);
6
7     while ( degree > 0 ) {
8         result *= num;
9         --degree;
10    }
11
12    return (n < 0) ? (1.0 / result) : result;
13 }
```

Функции в C++. Пример

```
1 double nth_degree(double num, int n)
2 { ... }
3
4 double rate = 4.5;
5 int scale = -4;
6
7 double rate_in_sc = nth_degree(rate, scale);
8 printf("rate^scale = %f\n", rate_in_sc);
9 printf("8.85^-3 = %f\n", nth_degree(8.85, -3));
```

Позиционные параметры

Параметры функций в C++ являются **позиционными** - при вызове функции аргументы подставляются в порядке, определяемом списком параметров.

Функции в C++. Передача аргументов по значению

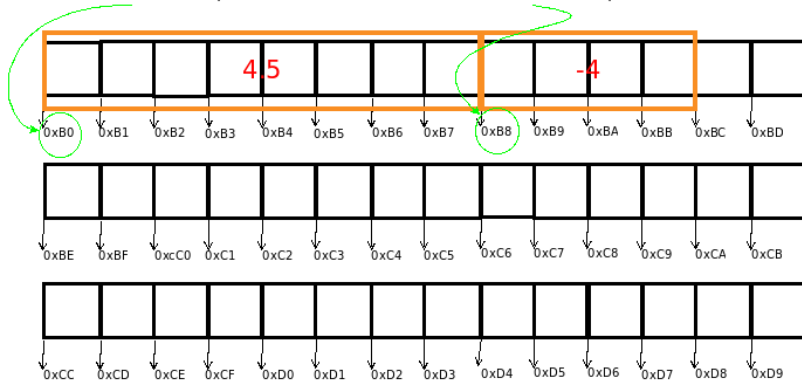
```
1 double nth_degree(double num, int n)
2 { ... }
3
4 double rate = 4.5;
5 int scale = -4;
6
7 double rate_in_sc = nth_degree(rate, scale);
```

Значения переменных **rate** и **scale** при вызове функции **degree_nth** называют её **аргументами**. В примере эти аргументы передаются в тело функции **по значению**.

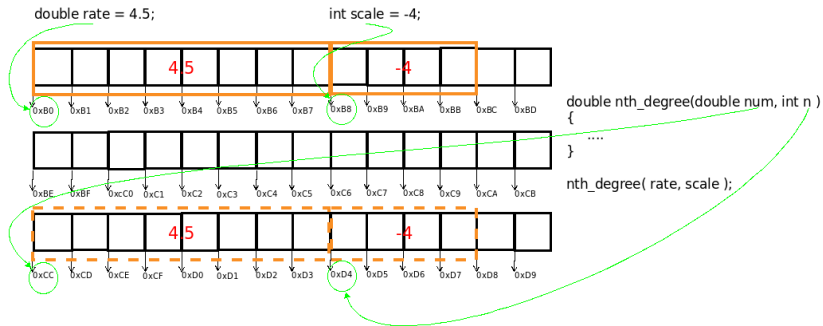
Функции в C++. Передача аргументов по значению

double rate = 4.5;

int scale = -4;



Функции в C++. Передача аргументов по значению

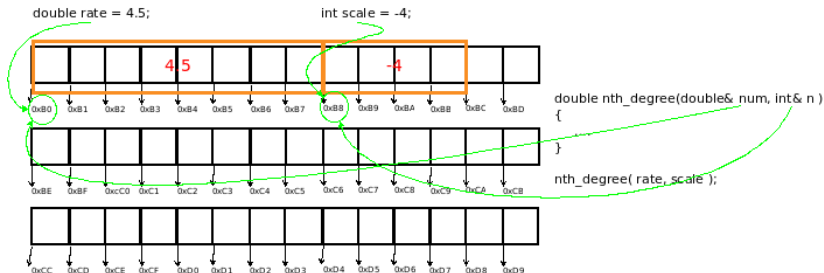


Функции в C++. Передача аргументов по ссылке

Есть второй способ передачи аргументов - **по ссылке**. Для этого в списке параметров *после* типа указывается знак «амперсанда» (&)

```
1 double nth_degree(double& num, int& n)
2 { ... }
3 ...
4
5 double rate = 4.5;
6 int scale = -4;
7
8 double rate_in_sc = nth_degree(rate, scale);
```


Функции в C++. Передача аргументов по ссылке



Важный факт. При передаче аргументов по ссылке, невозможно использовать литералы соответствующих типов (временные значения). То есть, невозможен вызов вида:

9 `nth_degree(3.5, 5);` // Ошибка компиляции

Функции в C++. Константные параметры

Параметры функции можно сделать **неизменяемыми (константными)**. В случае ссылочных параметров это позволяет передавать в функцию временные значения.

```
1 double degree_nth(const double& num,  
2                   const int& n)  
3 { ... }  
4  
5 ...  
6  
7 double rate = 4.5;  
8 int scale = -4;  
9  
10 double rate_in_scale = degree_nth( rate, scale );  
11 // Теперь можно и так:  
12 print("3.3^2 = ", degree_nth(3.3, 2), "\n");
```

Функции в C++. Пример

Пример: запросить у пользователя n целых чисел, найти максимальное и каким по счёту оно было введено

```
1 int find_max_with_pos(unsigned count, unsigned& place)
2 {
3     int current, max_num;
4     for (unsigned i = 1; i <= count; ++i) {
5         printf("Input %d number: ", i);
6         scanf("%d", &current);
7
8         if ( (i > 1) and (current > max_num) )
9             { max_num = current; place = i; }
10        else { max_num = current; place = 1; }
11    }
12    return max_num;
13 }
14 ...
15 unsigned position;
16 int max_elem = find_max_with_pos(5, position);
17 printf("Max number %d found at %u position\n",
18        max_elem, position);
```

Функции в C++. Параметры по умолчанию

```
1 double degree_nth(double num, int n = 3)
2 { ... }
3
4 ...
5
6 double rate = 4.5;
7
8 // Внутри функции *degree_nth* n равен 2
9 printf("%f\n", degree_nth( rate, 2 ));
10 // Выведет на экран: 20.25
11
12 // Внутри функции n равен 3
13 printf("%f\n", degree_nth( rate ));
14 // Выведет на экран: 91.125
```

Функции в C++. Параметры по умолчанию

1. Параметры со значением по умолчанию **должны** идти в конце списка.

```
1 double some_fun1(double r1, double r2,  
2                 double r3, int n1 = 2,  
3                 int n2 = 3)  
4 { ... }
```

2. Такие аргументы не могут быть **изменяемыми** **ссылочными** (константными ссылочными - могут).

```
1 // Ошибка компиляции  
2 double some_fun2(int& n1 = 2)  
3 { ... }  
4  
5 // Всё хорошо, компилятор доволен  
6 double some_fun3(const int& n1 = 2)  
7 { ... }
```

Функции в C++. Не хочу ничего возвращать

Возможно определять функции не требующие возврата значений из них. Для такого случая предусмотрен специальный тип данных - **void**

```
1 void print_only_even_number(const int num)
2 {
3     if ( (num % 2) != 0 ) {
4         return;
5     }
6
7     printf("%d\n", num);
8 }
9 // Переменных типа void создавать нельзя
10 // void var_of_void;
```

Функции в C++. Объявление и определение

Аналогично переменным, для функций существуют понятия **объявления** и **определения**. Определение - это все примеры выше, а под **объявлением** понимается указание *типа возвращаемого значения, названия функции и перечисление типов всех аргументов*. Причём, давать имена аргументам - не обязательно

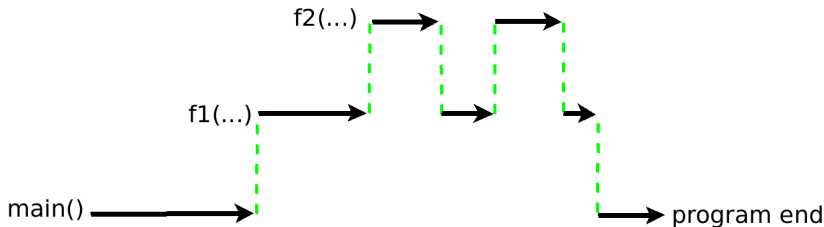
```
1 double my_rand(unsigned);           // Объявление
2
3 int main()
4 {
5     printf("Случайное число: %f\n", my_rand(5));
6 }
7
8 double my_rand(unsigned seed) // Определение
9 { return 1234.5688 / seed; }
```

Функции в C++. Поток выполнения

Под *потоком выполнения* будем понимать последовательное исполнение всех инструкций программы.

```
1 int f1(int); void f2(int& item);
2
3 int main()
4 {
5     int i1 = 44, i2 = f1(i1);
6     i2++;
7 }
8
9 int f1(int number)
10 {
11     number *= 10;    f2(number);
12     number /= 8.912; f2(number);
13     return 42;
14 }
15
16 void f2(int& item) { item++; }
```


Поток выполнения примера выше может быть представлен диаграммой:



Жизненный цикл программы на C++:

- текстовый файл с программой обрабатывается компилятором и создаётся *исполняемый файл* для ОС;
- исполняемый файл содержит инструкции для выполнения в двоичном виде (машинные коды);
- когда ОС загружает исполняемый файл — происходит загрузка двоичных инструкций в память ОС;
- инструкции начинают выполняться последовательно;
- каждая отдельная функция на C++ получает «свою» область в памяти ОС. Механизм выполнения функций называется **стеком вызовов**.

Прежде, чем идти дальше

Стек - специальная структура данных.

Прежде, чем идти дальше

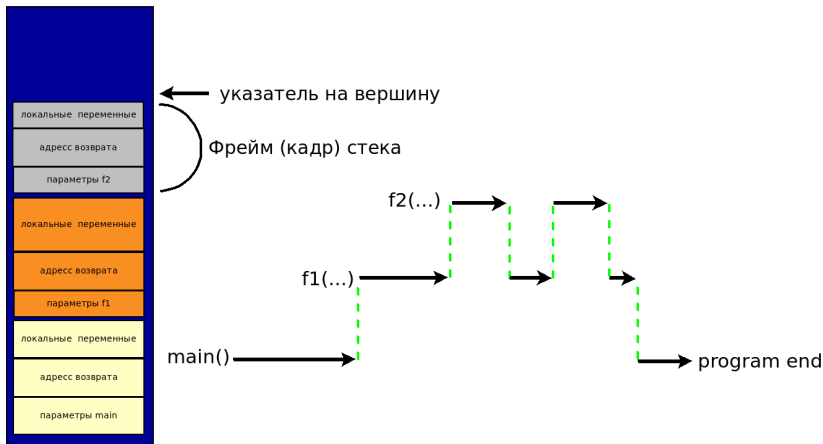
Стек - специальная структура данных.



Пример стека в жизни.

Функции в C++. Стек вызовов

Стек вызовов



Функции. Рекурсия



Функции. Рекурсия



В информатике под *рекурсией* понимается метод, в котором решение исходной задачи зависит от набора решений частных случаев этой же задачи. C++ для реализации подобных методов предоставляет возможность внутри функции делать вызов самой себя. Такие функции называются **рекурсивными**.

```
1 // Пример бесконечной (и хвостовой) рекурсии
2 int call_self(int num)
3 {
4     printf("Передано число: %d\n", num);
5     return call_self(num);
6 }
```


Функции. Рекурсия

Различают *прямую* и *косвенную* рекурсивную функцию.

- **прямая рекурсивная функция** - вызывает саму себя непосредственно в своём теле
- **косвенная рекурсивная функция** - вызывает некоторую другую функцию, в теле которой происходит обратный вызов исходной функции

```
1 int start(int);  
2  
3 int process_number(int num)  
4 {  
5     return start(num);  
6 }  
7  
8 int start(int n)  
9 {  
10     return process_number(n - 5);  
11 }
```

Для того, чтобы быть полезной, рекурсивная функция должна включать два необходимые особенности реализации:

- 1 Тело функции должно обязательно содержать **условия остановки рекурсивных вызовов**.
- 2 Параметры функции при рекурсивном вызове **должны меняться, а не оставаться постоянными**.

Числа Фибоначчи

$$F_1 = 1, F_2 = 1, F_n(n > 2) = F_{n-1} + F_{n-2}$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181, 6765, ...

Функции. Рекурсия

Числа Фибоначчи

$$F_0 = 0, F_1 = 1, F_2 = 1, F_n(n \geq 2) = F_{n-1} + F_{n-2}$$

```
1 unsigned long long fib_num(unsigned n)
2 {
3     // условие выхода из рекурсии
4     if (n < 2) {
5         return n;
6     }
7
8     return fib_num(n - 1) + fib_num(n - 2);
9 }
10
11 printf("8-ое число Фибоначчи: %llu\n",
12        fib_num(8));
```

- Каждая функция должна быть уникальна: определена в программе единожды
- Уникальность функции в языках программирования определяется её **сигнатурой** - набором отличительных признаков
- В C++ сигнатура функции определяется:
 - 1 Именем (идентификатором)
 - 2 Количеством параметров (термин *арность*)
 - 3 Типами параметров и их порядком

Функции в C++. Перегрузка

```
1 int max(const int first, const int second)
2 {
3     printf("max(int, int) works\n");
4     return (first > second) ? first : second;
5 }
6
7 double max(const double left, const double right)
8 {
9     printf("max(double, double) works\n");
10    const double EPS = 5e-5; // 5 * 10-5
11
12    if ( abs(left - right) > EPS ) {
13        return left;
14    } else {
15        return right
16    }
17 }
18
19 ...
20 printf("%f\n", max(56, 78));
21 printf("%f\n", max(3.4, 3.400007));
```

Пример из стандартной библиотеки C++

```
1  /*  
2     Получение модуля целого или  
3     действительного числа  
4  */  
5  #include <cmath>  
6  #include <cstdlib>  
7  
8  ...  
9  abs( 56 );  
10 abs( -8.888 );
```

Резюме по функциям

- Количество параметров выбираем произвольно, но вернуть можно только одно конкретное значение какого-нибудь типа
- Два способа передачи аргументов - **по значению** и **по ссылке**. В обоих случаях параметры можно сделать *константными*
- Части параметров можно присваивать значения по умолчанию
- Специальный тип **void** - когда из функции не нужно возвращать никакого значения
- Только название функции не уникально - помним о перегрузке
- Определение функции не может быть помещено в тело другой функции

Область видимости идентификатора

Место в файле с исходным кодом, где идентификатор доступен для операций (использование значения в случае переменной, вызов - в случае функции). В языке C++ различают две области - **глобальная** и **локальная**.

- Все функции имеют **глобальную** область видимости
- Переменная имеет **локальную** область видимости, если её объявление или определение находится внутри пары фигурных скобок {}
- Иначе переменная получает **глобальную** область видимости
- Локальная переменная видна во всех вложенных областях
- Идентификатор локальной переменной скрывает свои предыдущие определения

Прежде, чем идти дальше

Область видимости идентификатора

```
1 // Переменная с глобальной областью видимости
2 const double PI = 3.14159265358;
3
4 void do_something(double x)
5 {
6     // Локальная область видимости
7     double rate = 0.5;
8
9     if (x > 10.0) {
10         // Ещё одна. PI и rate тут доступны
11         double num = PI * x * rate;
12         print("Результат: ", num, "\n");
13     }
14
15     // Доступа к ним уже нет:
16     // num += 3.05;
17     rate *= 9.5;
18 }
```

Прежде, чем идти дальше

Время жизни переменной

- Переменные с локальной областью видимости автоматически удаляются по достижении конца области видимости
- Глобальные переменные существуют до тех пор, пока выполняется программа

Инициализация глобальных переменных

В отличие от локальных, глобальные переменные получают значения по умолчанию (или нулевые значения) в случае, если не сделана их инициализация. Для целочисленных типов данных это число «0», для действительных - «0.0», для логического - «false».