**Функции в С++**

Функции — это блоки кода, выполняющие определенные операции. Если требуется, функция может определять входные параметры, позволяющие вызывающим объектам передавать ей аргументы. При необходимости функция также может возвращать значение как выходное. Функции полезны для инкапсуляции основных операций в едином блоке, который может многократно использоваться. В идеальном случае имя этого блока должно четко описывать назначение функции. Следующая функция принимает два целых числа от вызывающего абонента и возвращает их сумму; *a* и *b* являются *параметрами* типа **int**.

int sum(int a, int b)

{

return a + b;

}

Функцию можно вызвать или *вызвать* из любого количества мест в программе. Значения, передаваемые функции, являются *аргументами*, типы которых должны быть совместимы с типами параметров в определении функции.

int main()

{

int i = sum(10, 32);

int j = sum(i, 66);

cout << "The value of j is" << j << endl; // 108

}

Длина функции практически не ограничена, однако для максимальной эффективности кода целесообразно использовать функции, каждая из которых выполняет **одиночную**, четко определенную задачу. Сложные алгоритмы лучше разбивать на более короткие и простые для понимания функции, если это возможно.

В C++, в отличие от других языков, функции можно также определять в области видимости пространства имен (включая неявное глобальное пространство имен). Такие функции называются *бесплатными* или *не-членными функциями*; они широко используются в стандартной библиотеке (std).

**Объявление функции**

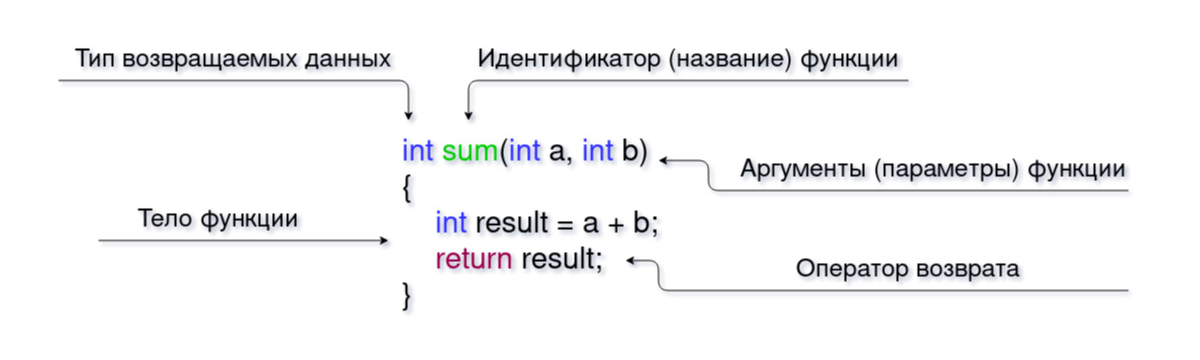
Минимальное *объявление* функции состоит из типа возвращаемого значения, имени функции и списка параметров (которые могут быть пустыми) вместе с необязательными ключевыми словами, предоставляющими дополнительные инструкции компилятору. В следующем примере показано объявление функции:

int sum(int a, int b);

Определение функции состоит из объявления, а также *тела*, который представляет собой весь код между фигурными скобками:

int sum(int a, int b)

{

 int result = a + b;

return result;

}

Объявление функции, за которым следует точка с запятой, может многократно встречаться в разных местах кода программы. Оно необходимо перед любыми вызовами этой функции в каждой записи преобразования. По правилу одного определения, определение функции должно фигурировать в коде программы лишь один раз.

**При объявлении функции необходимо указать:**

1. Тип возвращаемого значения, который указывает тип возвращаемого функцией значения или **void** если значение не возвращается. В C++11 является допустимым типом возвращаемого значения, **auto** который предписывает компилятору определить тип из инструкции return. В C++14 decltype(auto) также разрешено.
2. Имя функции, которое должно начинаться с буквы или символа подчеркивания и не должно содержать пробелов. В стандартной библиотеке со знака подчеркивания обычно начинаются имена закрытых функций-членов или функций, не являющихся членами и **не предназначенных** для использования в вашем коде.
3. Список параметров, заключенный в скобки. В этом списке через запятую указывается нужное (возможно, нулевое) число параметров, задающих тип и, при необходимости, локальное имя, по которому к значениям можно получить доступ в теле функции.

**Необязательные элементы объявления функции**

**constexpr** — указывает, что возвращаемое значение функции является константой, значение которой может быть определено во время компиляции.

constexpr float exp(float x, int n)

{

return n == 0 ? 1 :

n % 2 == 0 ? exp(x \* x, n / 2) :

exp(x \* x, (n - 1) / 2) \* x;

};

Спецификация компоновки **extern** или **static**.

//Declare printf with C linkage.

extern "C" int printf( const char \*fmt, ... );

**inline** — отдает компилятору команду заменять каждый вызов функции ее кодом. Подстановка может улучшить эффективность кода в сценариях, где функция выполняется быстро и многократно вызывается во фрагментах, являющихся критическими для производительности программы.

inline double Account::GetBalance()

{

return balance;

}

Выражение **noexcept** , указывающее, может ли функция вызывать исключение. В следующем примере функция не создает исключение, если is\_pod выражение принимает значение **true**.

#include <type\_traits>

template <typename T>

T copy\_object(T& obj) noexcept(std::is\_pod<T>) {...}

***\****А также:

* (Только функции-члены) Квалификаторы cv, указывающие, является **const** ли функция или **volatile**.
* (Только функции-члены) **virtual**или **finaloverride**. **virtual** — указывает, что функция может быть переопределена в производном классе. **override** — означает, что функция в производном классе переопределяет виртуальную функцию. **final** — означает, что функция не может быть переопределена ни в одном из последующих производных классов.
* (Только функции-члены) **static** к функции-члену означает, что функция не связана ни с какими экземплярами объекта класса.
* (Только нестатические функции-члены) Квалификатор ссылок, указывающий компилятору, какой перегрузке функции выбирать, когда неявный параметр объекта (\*this) является ссылкой rvalue и lvalue.

**Определение функции**

*Определение функции* состоит из объявления и тела функции, заключенных в фигурные скобки, которые содержат объявления переменных, операторы и выражения. В следующем примере показано полное определение функции:

int foo(int i, std::string s)

{

int value {i};

MyClass mc;

if(strcmp(s, "default") != 0)

{

value = mc.do\_something(i);

}

return value;

}

***\****Переменные, объявленные в теле функции, называются локальными. Они исчезают из области видимости при выходе из функции, поэтому функция никогда не должна возвращать ссылку на локальную переменную.

MyClass& boom(int i, std::string s)

{

int value {i};

MyClass mc;

mc.Initialize(i,s);

return mc; }

**Создание функций внутри одного файла**

#include <iostream>

//Новые функции и процедуры должны создаваться выше функции main()

//В круглых скобках указываются аргументы, которые будет принимать функция

int sum(int a, int b)

{

//Используем переданные аргументы, чтобы провести вычисления

int result = a + b;

//Оператор возврата говорит программе, что функция завершила работу

return result;

//Всё, что записано после return, не будет выполнено

}

int main()

{

//Вызываем функцию с аргументами

int result = sum(5, 6);

}

Функции могут принимать или не принимать аргументы. Но если вы указали, что аргументы всё же нужны, то попытка вызвать функцию без них приведёт к ошибке.

Если вы указываете какой-либо тип возвращаемых данных, то обязательно должен присутствовать оператор *return*. Если возвращать значение не нужно, создайте процедуру — укажите тип *void*.

Функция *main ()* автоматически возвращает значение, поэтому отдельно прописывать *return* не обязательно. Программа вернёт 0, если она выполнилась успешно, или другое число в зависимости от ошибки.

**Области видимости переменных в функциях**

Всё, что происходит в функции, в ней же и остаётся. То есть функции могут работать только с теми значениями, которые в них передали:

#include <iostream>

void printHello()

{

//Переменная name объявлена в другой функции, поэтому у printHello() нет к ней доступа

//Запуск такого кода приведёт к ошибке

std::cout << "Hello, " << name << "!\n";

}

int main()

{

std::string name = "Igor";

printHello();

}

Если переменная создаётся внутри какого-либо блока {}, то она будет доступна только в этом и во всех вложенных блоках; такие переменные называются **локальными.**

Если вы хотите, чтобы какая-нибудь переменная была доступна везде, то её нужно объявить за пределами каких-либо блоков; такие переменные называются **глобальными:**

#include <iostream>

//Глобальные переменные создаются вне функций

std::string name;

void printHello()

{

std::cout << "Hello, " << name << "!\n";

}

int main()

{

name = "Igor";

printHello();

}

Однако использовать глобальные переменные не рекомендуется, потому что вам будет сложно отслеживать изменения и влиять на результат работы программы. Единственное исключение — это константы, которые нужно делать глобальными, потому что они не меняются.

**Аргументы функции в С++**

Рассмотрим, что происходит, когда вы передаёте аргумент в функцию. Допустим, есть вот такой код:

#include <iostream>

void sum(int a)

{

a = a + 500;

}

int main()

{

int a = 5;

sum(a);

std::cout << a << "\n";

}

Можно подумать, что в результате программы мы увидим число 505, но это неверно. Дело в том, что функции в качестве аргументов принимают не сами переменные, а их значения. То есть в *sum ()* попала не ячейка памяти, а число 5, которое находилось в этой ячейке. Учитывайте это, когда пишете программы.

Это касается только примитивных типов данных. Ссылочные типы ведут себя иначе.

**Аргументы по умолчанию**

Функция может принимать аргументы по умолчанию, то есть некоторые значения, которые функция использует, если при вызове для параметров явным образом не передается значение:

#include <iostream>

void multiply(int n, int m = 3)

{

int result = n \* m;

std::cout << "n \* m = " << result << std::endl; }

int main() {

multiply(4, 5);

multiply(4);

return 0;

}

Для установки значения по умолчанию параметру присваивается это значение int m = 3. И если для второго параметра не будет передано значение, то он будет использовать значение по умолчанию.

При объявлении прототипа подобной функции он тоже может содержать значение по умолчанию для параметра. И в этом случае мы можем не определять в функции значение по умолчанию для параметра - оно будет браться из прототипа:

#include <iostream>

void multiply(int n, int m=3);

int main()

{

multiply(4, 5);

multiply(4);

return 0;

}

void multiply(int n, int m)

{

int result = n \* m;

std::cout << "n \* m = " << result << std::endl;

}

**Передача аргументов по значению**

Аргументы могут передаваться по значению (by value) и по ссылке (by reference). При передаче аргументов по значению внешний объект, который передается в качестве аргумента в функцию, не может быть изменен в этой функции. В функцию передается само значение этого объекта. Например:

#include <iostream>

void square(int, int);

int main()

{

int a = 4;

int b = 5;

std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;

square(a, b);

std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

return 0;

}

void square(int a, int b)

{

a = a \* a;

b = b \* b;

std::cout << "In square: a = " << a << "\tb=" << b << std::endl;

}

Функция square принимает два числа и возводит их в квадрат. В функции main перед и после выполнения функции square происходит вывод на консоль значений переменных a и b, которые передаются в square в качестве аргументов.

И при выполнении мы увидим, что изменения аргументов в функции square действуют только в рамках этой функции. Вне ее значения переменных a и b остаются неизменными.

Так происходит т.к. при компиляции функции для ее параметров выделяются отдельные участки памяти. При вызове функции вычисляются значения аргументов, которые передаются на место параметров. И затем значения аргументов заносятся в эти участки памяти. То есть функция манипулирует копиями значений объектов, а не самими объектами.

**Передача параметров по ссылке**

При передаче параметров по ссылке передается ссылка на объект, через которую мы можем манипулировать самим объектов, а не просто его значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

#include <iostream>

void square(int&, int&);

int main()

{

int a = 4;

int b = 5;

std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;

square(a, b);

std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

return 0;

}

void square(int &a, int &b)

{

a = a \* a;

b = b \* b;

std::cout << "In square: a = " << a << "\tb=" << b << std::endl;

}

Теперь параметры a и b передаются по ссылке. Ссылочный параметр связывается непосредственно с объектом, поэтому через ссылку можно менять сам объект.

И если мы скомпилируем и запустим программу, то результат будет иным.

Передача по ссылке позволяет возвратить из функции сразу несколько значений. Также передача параметров по ссылке является более эффективной при передаче очень больших объектов. Поскольку в этом случае не происходит копирования значений, а функция использует сам объект, а не его значение.

От передачи аргументов по ссылке следует отличать передачу ссылок в качестве аргументов:

#include <iostream>

void square(int, int);

int main()

{

int a = 4;

int b = 5;

int &aRef = a;

int &bRef = b;

std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;

square(aRef, bRef);

std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

return 0;

}

void square(int a, int b)

{

a = a \* a;

b = b \* b;

std::cout << "In square: a = " << a << "\tb=" << b << std::endl;

}

Если функция принимает аргументы по значению, то изменение параметров внутри функции также никак не скажется на внешних объектах, даже если при вызове функции в нее передаются ссылки на объекты.

**Передача массивов в функции**

Из-за того, что копирование больших массивов при передаче в функцию является очень затратной операцией, C++ не копирует массив. При передаче массива в качестве аргумента в функцию, массив распадается в указатель на массив и этот указатель передается в функцию:

#include <iostream>

void printSize(int \*array)

{

// Здесь массив рассматривается как указатель

std::cout << sizeof(array) << '\n'; // выведется размер указателя, а не длина массива!

}

int main()

{

int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };

std::cout << sizeof(array) << '\n'; // выведется sizeof(int) \* длина массива

printSize(array); // здесь аргумент array распадается на указатель

return 0;

}

Результат выполнения программы:

32   
 4

Обратите внимание, результат будет таким же, даже если параметром будет фиксированный массив:

#include <iostream>

// C++ неявно конвертирует параметр array[] в \*array

void printSize(int array[])

{

// Здесь массив рассматривается как указатель, а не как фиксированный массив

std::cout << sizeof(array) << '\n'; // выведется размер указателя, а не размер массива!

}

int main()

{

int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };

std::cout << sizeof(array) << '\n'; // выведется sizeof(int) \* длина массива array

printSize(array); // здесь аргумент array распадается на указатель

return 0;

}

Результат выполнения программы:

32   
 4

**Итог:**

Передача параметров по значению больше подходит для передачи в функцию небольших объектов, значения которых копируются в определенные участки памяти, которые потом использует функция.

Передача параметров по ссылке больше подходит для передачи в функцию больших объектов, в этом случае не нужно копировать все содержимое объекта в участок памяти, за счет чего увеличивается производительность программы.

**Константные параметры**

Параметры могут быть константными - значения таких параметров не могут меняться. Например:

void square(const int n)

{

// n = n \* n; // можно считать значение параметра, но не изменять его

std::cout << n \* n << std::endl;

}

Константному параметру можно передать в качестве аргумента как константу, так и переменную.

Следует помнить, что при передаче параметров по значению, мы делаем копию значения переданной переменной, а это значит, что переданные константы в не константные параметры могут меняться.

#include <iostream>

void square(int, int);

int main()

{

const int a = 4;

const int b = 5;

square(a, b); // 400

return 0;

}

void square(int a, int b)

{

a = a \* a;

b = b \* b;

std::cout << "In square: a \* b = " << a \* b << std::endl;

}

**Константные ссылки**

Если функция получает аргументы по ссылке, то чтобы передать в функцию константу, параметры тоже должны представлять ссылку на константу:

#include <iostream>

void square(const int&, const int&);

int main()

{

const int a = 4;

const int b = 5;

square(a, b); // 20

return 0; }

void square(const int &a, const int &b)

{

// a = a \* a; так нельзя сделать

// b = b \* b; так нельзя сделать

std::cout << "In square: a \* b = " << a \* b << std::endl; }

И если в функцию необходимо передать большие объекты, которые не должны изменяться, то определение параметров именно как константных ссылок больше всего подходит для данной задачи.

**Возвращение ссылки**

Не следует возвращать ссылку на локальный объект, который создается внутри функции. Поскольку все создаваемые в функции объекты удаляются после ее завершения, а их память очищается, то возвращаемая ссыла будет указывать на несуществующий объект, как в следующем случае:

int &factorial(int n)

{

int result = 1;

for(int i = 1; i <=n; i++)

{

result \*= i; }

return result; }

**Массивы в параметрах функции**

Если функция принимает в качестве параметра массив, то фактически в эту функцию передается указатель на первый элемент массива. То есть как и в случае с указателями нам доступен адрес, по которому мы можем менять значения. Поэтому следующие объявления функции будут по сути равноценны:

void print(int numbers[]);

void print(int \*numbers);

Передадим в функцию массив:

#include <iostream>

void print(int[]);

int main()

{

int nums[] = {1, 2, 3, 4, 5};

print(nums);

return 0; }

void print(int numbers[])

{

std::cout << "First number: " << numbers[0] << std::endl; }

В данном случае функция print выводит на консоль первый элемент массива.

Теперь определим параметр как указатель:

#include <iostream>

void print(int\*);

int main()

{

int nums[] = {1, 2, 3, 4, 5};

print(nums);

return 0; }

void print(int \*numbers)

{

std::cout << "First number: " << \*numbers << std::endl;}

Здесь также в функцию передается массив, однако параметр представляет указатель на первый элемент массива.

**Ограничения**

Поскольку параметр, определенный как массив, рассматривается именно как указатель на первый элемент, то мы не сможем корректно получить длину массива, например, следующим образом:

void print(int numbers[])

{

int size = sizeof(numbers) / sizeof(numbers[0]);

std::cout << size << std::endl;

}

И также мы не сможем использовать цикл for для перебора этого массива:

void print(int numbers[])

{

for (int n : numbers)

std::cout << n << std::endl;

}

**Передача маркера конца массива**

Чтобы должным образом определять конец массив, перебирать элементы массива, необходимо использовать специальный маркер, который бы сигнализировал об окончании массива. Для этого могут использоваться разные подходы.

Первый подход заключается в том, чтобы один из элементов массива сам сигнализировал о его окончании. В частности, массив символов может представлять строку - набор символов, который завершается нулевым символом '\0'. Фактически нулевой символ служит признаком окончания символьного массива:

#include <iostream>

void print(char[]);

int main()

{

char chars[] = "Hello";

print(chars);

return 0;

}

void print(char chars[])

{

for (int i = 0; chars[i] != '\0'; i++)

{

std::cout << chars[i] << "\t";

}

}

Второй подход заключается в передаче в функцию размера массива.

Третий подход заключается в передаче указателя на конец массива. Можно вручную вычислять вычислять указатель на конец массива. А можно использовать встроенные библиотечные функции std::begin() и std::end():

int nums[] = { 1, 2, 3, 4, 5 };

int \*begin = std::begin(nums); // указатель на начало массива

int \*end = std::end(nums); // указатель на конец массива

Причем end возвращает указатель не на последний элемент, а адрес за последним элементом в массиве.

**Константные массивы**

Поскольку при передаче массива передается фактически указатель на первый элемент, то используя этот указатель, мы можем изменить элементы массива. Если нет необходимости в изменении массива, то лучше параметр-массив определять как константный:

#include <iostream>

void print(const int\*, const int\*);

void twice(int\*, int\*);

int main()

{

int nums1[] = { 1, 2, 3, 4, 5 };

int \*begin = std::begin(nums1);

int \*end = std::end(nums1);

print(begin, end);

std::cout << std::endl;

int nums2[] = { 1, 2, 3, 4, 5 };

begin = std::begin(nums2);

end = std::end(nums2);

twice(begin, end);

for (int \*ptr = begin; ptr != end; ptr++)

{

std::cout << \*ptr << "\t";

}

std::cout << std::endl;

return 0;

}

void print(const int \*begin, const int \*end)

{

for (const int \*ptr = begin; ptr != end; ptr++)

{

std::cout << \*ptr << "\t";

}

}

void twice(int \*begin, int \*end)

{

for (int \*ptr = begin; ptr != end; ptr++)

{

\*ptr = \*ptr \* 2;

}

}

В данном случае функция print просто выводит значения из массива, поэтому параметры этой функции помечаются как константные.

Функция twice изменяет элементы массива - увеличивает их в два раза, поэтому в этой функции параметры являются не константными. Причем поле выполнения функции twice массив nums3 будет изменен.

**Передача многомерного массива**

Статический многомерный массив передаётся идентично одномерному. Но многие испытывают проблему при передаче данных многомерного динамического массива в функцию, поэтому разберём этот момент подробнее на более сложном примере***\****.

Поскольку многомерный динамический массив реализуется как одномерный массив указателей, то, соответственно, и при работе с функциями применяются те же подходы, что и для одномерного массива.

Для примера — полный код программы, демонстрирующей работу с двумерным динамическим массивом с использованием функций:

#include <iostream>

#include <iomanip>

using namespace std;

const unsigned int DIM1 = 3;

const unsigned int DIM2 = 5;

int \*\*array\_generator(unsigned int dim1, unsigned int dim2) {

int \*\*ptrary = new int \* [dim1];

for (int i = 0; i < dim1; i++) {

ptrary[i] = new int [dim2];

}

return ptrary;

}

void array\_destroyer(int \*\*ary, unsigned int dim1) {

for (int i = 0; i < dim1; i++) {

delete [] ary[i];

}

delete [] ary;

}

int main() {

int \*\*matrix;

// создание массива

matrix = array\_generator(DIM1, DIM2);

// использование

for (int i = 0; i < DIM1; i++) {

for (int j = 0; j < DIM2; j++) {

matrix[i][j] = (i + 1) \* 10 + (j + 1);

cout << setw(4) << matrix[i][j];

}

}

// уничтожение

array\_destroyer(matrix, DIM1);

return 0;

}

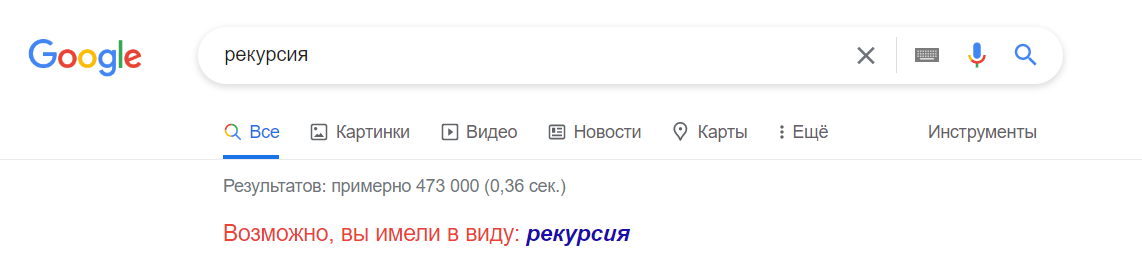
***\****Скопируйте код программы и разберите, что делает данный алгоритм.

**Рекурсивный вызов функций**

Рекурсия — это ситуация, когда объект является частью самого себя *(не путать с фракталом)*. В программировании это проявляется в одной интересной особенности функций — они могут вызывать сами себя.

*Чтобы понять, что такое рекурсия, нужно понять, что такое рекурсия*

Чаще всего рекурсия объясняется на примере факториалов*(не путать с фракталами):*

#include <iostream>

int factorial(int);

int main()

{

int n = 5;

int result = factorial(n);

std::cout << "Factorial of " << n << " is equal to " << result << std::endl;

return 0;

}

int factorial(int n)

{

if(n>1)

return n \* factorial(n-1);

return 1; }

В функции factorial задано условие, что если число n больше 1, то это число умножается на результат этой же функции, в которую в качестве параметра передается число n-1. То есть происходит рекурсивный спуск. И так далее, пока не дойдем того момента, когда значение параметра не будет равно 1. В этом случае функция возвратит 1.

Рекурсивная функция обязательно должна иметь некоторый базовый вариант, который использует оператор return и к которому сходится выполнение остальных вызовов этой функции. В случае с факториалом базовый вариант представлен ситуацией, при которой n = 1. В этом случае сработает инструкция return 1;. Иначе программа попадёт в бесконечный цикл и сильно нагрузит компьютер.

На практике применение этой особенности можно увидеть при рекурсивном удалении папок. Функция удаления должна работать так:

1. Сканирует указанную папку.
2. Удаляет все файлы из этой папки.
3. Сканирует папки внутри этой папки и выполняет предыдущие шаги для каждой из них.