

I

# Изучить C++ – это просто

Дни 1-10

Изучаете переменные, константы, массивы, строки, выражения, операторы, функции...



Дни 11-21

Изучаете процесс выполнения программы, указатели, ссылки, классы, объекты, наследование, полиморфизм...



Дни 22-697

Много программируете для развлечения. Получаете удовольствие, занимаясь хакерством, но не забываете учиться на своих ошибках.



Дни 698-3648

Общаетесь с другими программистами, вместе работаете над программными проектами, учитесь у них.



Дни 3649-7781

Изучаете высшую теоретическую физику и формулируете непротиворечивую теорию квантовой гравитации.



Дни 7782-14611

Изучаете биохимию, молекулярную биологию, генетику...



День 14611

Основываясь на своих знаниях биологии, создаете омолаживающее снадобье.



День 14611

Основываясь на своих знаниях физики, создаете потоковый накопитель и возвращаетесь в прошлое - в день 21.



День 21

Заменяете молодого себя.



Насколько мне известно, это самый легкий способ «Выучить C++ за 21 день».

Для их создания могут быть использованы два ключевых слова — **using** или **typedef**. Первое из них появилось со стандарта C++11 (конец 2011 года) и на данный момент является предпочтительным способом для определения псевдонимов, второе — пришло из языка C, долгое время использовалось и в C++, и теперь уже осталось для совместимости с ранее написанным кодом.

# Отличия от C: псевдонимы

В актуальный C++ добавление псевдонимов осуществляется с помощью ключевого слова **using**. Общий синтаксис следующий:

`using <псевдоним> = <тип_данных>;`

```
1 using ull_t      = unsigned long long;
2 using ull_ptr_t = unsigned long long*;
3 using real_fn_t = double (*)(double);
4
5 ull_t      val1 = 5555555555;
6 ull_ptr_t ptr1 = &val1;
7 printf("ptr pointed to %Lu\n", *ptr1);
8
9 real_fn_t sin_fn = sin;
10 printf("sin(Pi / 2) = %f\n", sin_fn(3.14 / 2));
```

Главное отличие от **typedef**: каждому псевдониму — отдельное объявление.

Для сравнения

```
1 typedef unsigned long long ull_t, *ull_ptr_t;  
2 typedef double (*real_fn_t)(double);  
3  
4 ...
```

## Отличия от C: псевдонимы

Предметно-ориентированность: тригонометрические функции, принимающие значения в градусах, а не радианах.

```
1 using angle_t = double;
2
3 double sin_at(angle_t value)
4 { return sin(value * M_PI / 180); }
5
6 double cos_at(angle_t value)
7 { return cos(value * M_PI / 180); }
8
9 double tg_at(angle_t value)
10 { return tan(value * M_PI / 180); }
11
12 printf("tg(45 degree) = %.3f\n", tg_at(45.0));
```

- 1 Параметры: значения по умолчанию

$$\int_a^b f(x) dx$$

```
1 using real_fn_t = double (*)(double);
2
3 double integrate1D(real_fn_t fn,
4     double a, double b, double eps=0.0001);
5
6 ...
7
8 double value1 = integrate1D(sin, 0, 3.14/4);
9 double value2 = integrate1D(exp, 1.27, 5.14,
10     0.000001);
```

## 2 Перегрузка функций

### Термин

**Сигнатура** функции – набор отличительных признаков, обеспечивающий уникальность функции в текущей программе.

В C++ сигнатура функции определяется:

- 1 Именем (идентификатором)
- 2 Количеством параметров (термин *арность*)
- 3 Типами параметров и их порядком

```
1 void cat_shows(int value)
2 {
3     printf("  /\_\_/\ \ \n"
4           "  ( o.o ) ->  %d (int)\n"
5           "  > ^ < / \n", value);
6 }
```



# Отличия от C: функции II

```
7
8 void cat_shows(double value)
9 {
10     printf("  /\\_/\ \n"
11           " ( o.o ) ->  %+.4f (double)\n"
12           "  > ^ < / \n", value);
13 }
14
15 ...
16
17 cat_shows(-5);
18 cat_shows(4.5);
```

## Термин

**Литерал** – символьное обозначение значения некоторого типа данных в программе.

# Отличия от C: нулевой указатель

Вместо макроса **NULL** используется ключевое слово **nullptr**.  
Поиск первого вхождения символа в строке

```
1 #include <cstring>
2 char text[] = "That's me in the corner\n"
3               "That's me in the spotlight\n"
4               "Losing my religion\n"
5               "Trying to keep up with you\n"
6               "And I don't know if I can do it\n"
7               "Oh no, I've said too much\n"
8               "I haven't said enough\n";
9 printf("Find all apostrophes in\n%s\n", text);
10 puts("-----");
11 const char apostr = '\'';
12
13 char *symb_ptr = strchr(text, apostr);
14 while (symb_ptr != nullptr) {
15     int place = symb_ptr - text + 1;
16     printf("%c found at %d place\n", apostr, place);
17     symb_ptr = strchr(symb_ptr + 1, apostr);
18 }
```

# Отличия от C: инициализация

- (1) `Type var{};`  
`Type var();`
- (2) `Type var;`
- (3) `Type var{val1, [, val2, ...]};`  
`Type var(val1, [, val2, ...]);`
- (4) `Type var = value;`  
`Type var = {val1, [, val2, ...]};`

- ❶ Инициализация значением по умолчанию (value initialization). Для фундаментальных типов и указателей на любые типы значением по умолчанию является **нуль**.

```
1 int i_var{};
2 double real{};
3 int my_array[12]{};
4
5 printf("%d, %d, %d\n", i_var == 0, real == 0.0,
6        my_array[1] == 0 && my_array[4] == 0.0);
```

- ② Инициализация по умолчанию (default initialization).  
Различна для **локальной** и **глобальной** областей видимости. В локальной области переменная получает произвольное значение, в глобальной — будет проинициализированна **значением по умолчанию**.

```
1 int gl_count;  
2 int *ptr;  
3  
4 int main()  
5 {  
6     int count;  
7     printf("Global default: %d, local: %d\n",  
8           gl_count, count);  
9     printf("has pointer zero-value: %d\n",  
10          ptr == nullptr);  
11 }
```

# Отличия от C: инициализация

- 3 Прямая инициализация (direct initialization).
- 4 Инициализация копированием (copy initialization).  
Присваивает переменной конкретное значение. Для элементов составных объектов, которым не предоставлено значения, применяется **инициализация значением по умолчанию**

```
1 struct Pair
2 { int first, second; };
3
4 int views{1000}, likes = 5, votes = {6};
5 double real_10th[10] = {1.0, 3.0, 5.0};
6
7 Pair p1{4, -8};
8 Pair p2{-11};
9
10 print("p2.second = %d\n", p2.second);
```

## Отличия от C: динамическая память

Оператор **new** - запрос динамической памяти у ОС на один объект заданного типа. Размер выделенного блока совпадает с размером типа данных. При успешном выполнении, оператор возвращает **адрес выделенного блока**.

```
new <тип>{<значение_для_инициализации>} ;
```

```
new <тип>(<значение_для_инициализации>) ;
```

```
1 int *p1 = new int; // инициализация по умолчанию
2 *p1 = 89;
3 printf("Value of dynamic int obj is %d\n", *p1);
4
5 // инициализация значением по умолчанию
6 int *p2 = new int{};
7 printf("Value pointed by p2 is %d\n", *p2);
8
9 int *p3 = new int{101}; // прямая инициализация
10 bool is_101 = *p3 == 101;
```

При ошибке выделения: завершение работы программы

## Отличия от C: динамическая память

Оператор **delete** - возвращение блока динамической памяти обратно ОС, выделенной под один объект конкретного типа

`delete <переменная-указатель>;`

```
1 int *p1 = new int;  
2 *p1 = 89;  
3 delete p1;  
4 // !Недопустимо, ошибка времени выполнения!  
5 //delete p1;  
6  
7 p1 = nullptr;  
8 delete p1; // Безопасно  
9 delete p1; // Сколь угодно раз подряд
```

**Правило хорошего тона:** для указателя на динамический блок памяти **обязателен** вызов оператора **delete**.

# Отличия от C: динамическая память

Оператор **new**[]): выделение блока динамической памяти под массив заданного размера конкретного типа

`new <тип>[<размер_массива>] {<инициализация>;`

```
1 size_t elems_count;
2 printf("Enter array size: ");
3 scanf("%zu", &elems_count);
4
5 int *dyn_array = new int[elems_count];
6 for (size_t i = 0; i < elems_count; i++) {
7     dyn_array[i] = i + 1;
8 }
```

Выделяется массив под заданное с консоли количество элементов, к массиву применяется *инициализация по умолчанию*. Затем происходит работа с ним.



# Отличия от С: динамическая память

Оператор **new[]**: инициализация массивов начальными значениями

```
1 int *arr1 = new int[10]{};
2 bool is_zero = p1[0] == 0;
3 // is_zero здесь равен true
4
5 double *arr2 = new double[12] {1.0, 2.0, 3.0};
6 is_zero = arr2[4] == 0.0;
7 // опять
```

# Отличия от C: динамическая память

Оператор **delete[]**: возвращение памяти, выделенной под массив

`delete[] <переменная-указатель>;`

```
1 int *my_ints = new int[10];
2
3 my_ints[2] = 98;
4 my_ints[4] = 89;
5
6 printf("Sum of third and fifth is %d\n",
7       my_ints[2] + my_ints[4]);
8
9 delete[] my_ints;
10 // delete[] my_ints; // !Так не делать!
```

**Повторение:** безопасно применять операторы **delete/delete[]** можно только к указателю, который хранит *нулевой адрес* (значение **nullptr**).

- Явное приведение значений числовых фундаментальных типов

```
1 int i_val = 345;
2 double r_val = -18.4567;
3 // C style
4 int int_part = (int) r_val;
5
6 // C++ style
7 // Option 1
8 int another = int(r_val);
9 double real = double(i_val);
10 // Option 2
11 int value = static_cast<int>(578.934);
```

# Отличия от C: приведение типов

- Приведение указателей

```
1 double value = 4.55, *real_ptr = &value;  
2 // C style  
3 void *ptr1 = (void*) real_ptr;  
4  
5 // C++ style  
6 char *ptr2 = reinterpret_cast<char*>(&value);
```

# Пространства имён (**namespaces**)

В C++ любое **объявление** переменной, функции или пользовательских типов данных может быть помещено в **пространство имён**.

Технически, **пространство имён** - это **лексическая** область видимости для группы *идентификаторов*.

По смыслу, **пространство имён** - это именованное множество, название которого необходимо для получения доступа к определённому идентификатору переменной/функции/типа.

**Пространства имён в C++ открыты для расширения:** в любое из них (хоть из стандартной библиотеки, хоть из собственной, хоть из внешней) каждая программа может добавить свой набор переменных/функций/типов.

**Пространство имён** создаётся с помощью ключевого слова **namespace** и выбора названия. Например,

```
1 namespace fp_omsu
2 {
3
4 const size_t PLAYERS = 7;
5
6 bool is_same(int left, int right)
7 {
8     return left == right;
9 }
10
11 }
```

Имя пространства имён - **fp\_omsu**, внутри него содержатся — одна константа и одна функция.

# C++: пространства имён

Само пространство имён ограничено блоком из фигурных скобок (строки 2 и 10 с предыдущего слайда).

Ко всему содержимому пространства имён можно обратиться с помощью его имени и оператора **двойного двоеточия** (разрешение области видимости). Для примера:

```
18 printf("PLAYERS: %d\n", fp_omsu::PLAYERS);
19
20 int i1 = 12, i2 = 12;
21 bool same = fp_omsu::is_same(i1, i2);
22 printf("i1 and i2 are same? %d\n", same);
```

Всё как обычно, разве что для всех сущностей появилась постоянная приставка «**fp\_omsu::**».



Для получения доступа ко **всем** идентификаторам используется ключевое слово **using**.

```
1 using namespace fp_omsu;
```

- данное использование **using** делает **все идентификаторы** из пространства имён **fp\_omsu** доступными в текущей **области видимости** (другими словами, происходит *импорт имён*).  
Возможен частичный импорт идентификаторов:

```
1 using fp_omsu::is_same;  
2 // Функция is_same становится доступна по имени  
3 bool status = is_same(5, 7);  
4 // Но не константа PLAYERS  
5 size_t extended = fp_omsu::PLAYERS + 5;
```

# C++: пространства имён

Вся стандартная библиотека C++ заключается в пространство имён **std**. Вследствии чего, во многих примерах книг и на просторах интернета появляется строка:

```
1 using namespace std;
```

При этом, часть стандартной библиотеки языка C, доступная в C++ — не следует данному правилу. Поэтому, для функций **printf** или **scanf** не требуется указание префикса «**std::**».

Пример того, как её отсутствие могло повлиять на имена функций:

```
1 #include <cmath>
2 #include <cstdlib>
3
4 abs( 56 );
5 std::abs( -8.888 );
```

Без включения пространства имён **std**, функция получения модуля для типов, представляющих числа с плавающей точкой, требует явной записи своего полного названия.

## Правило хорошего тона

В современном C++ рекомендуется по возможности использовать полный импорт идентификаторов (**using namespace** **fp\_omsu** и ему подобные) только внутри отдельных блоков кода, ограниченных парой *фигурных скобок* (функции, структуры, классы, другие пространства имён).

## Зачем вообще нужны?

Пространства имён позволяют использовать разные библиотеки, содержащие одинаковые по именованию сущности. Например, если есть функция **sort** в библиотеках **lib1** и **lib2**, то в своей программе можно использовать обе реализации, если внутри библиотек используются разные пространства имён. И у компилятора не будет никаких претензий (т.е. **lib1::sort**, **lib2::sort**).

## Ссылочные типы (references)

**Ссылки** в C++ — специальный тип данных, позволяющий получать не прямой доступ к значению некоторого объекта. В C++ есть два вида ссылок. В целях упрощения изложения материала и терминологии, будем различать их как **ссылки на переменные** и **ссылки на временные объекты**.

Примеры временных объектов для различных типов:

```
1 4;    // литерал типа int
2 3.25; // литерал типа double
3
4 // Ниже всё, что в фигурных скобках,
5 // является литералом составного типа.
6 // В данном случае, этот тип – массив int'ов
7 // на 4 элемента.
8 int arr[] = {3, 4, 5, 6};
9
10 "abcde"; // литерал типа const char*
```

В строках **1, 2, 8, 10** присутствуют *временные объекты* (также употребляется термин «временные значения»)

Возвращаясь к ссылкам, общий вид создания *переменных-ссылок* для обоих случаев следующий:

```
// ссылка на переменную  
Type & ref_name = var;
```

```
// ссылка на временный объект  
Type && ref_name = temp_var;
```

Знаки одного (**&**) и двух (**&&**) амперсандов говорят о том, что создаются ссылки каждого вида на тип **Type**. Кроме того, знак присваивания и некоторый объект являются обязательными. Под **var** понимается какая-то переменная соответствующего типа, под **temp\_var** — временный объект. Ссылки в C++ не имеют никакого **нулевого значения**, они всегда должны быть связаны с каким-нибудь объектом программы. Аналогично указателям, ссылки являются *типизированными*.

Для начала, пример с **ссылками на переменные**:

```
1 int i_num = 293;
2 double real = 55.88;
3
4 int &i_ref = i_num;
5 double &r_ref = real;
6 // вычитаем 13 и сохраняем результат в i_num
7 i_ref -= 13;
8 printf("i_num = %d; i_ref = %d\n",
9        i_num, i_ref);
10 // удаляем значение переменной real
11 r_ref *= 2.0;
12 printf("real = %.3f; r_ref = %.3f\n",
13        real, r_ref);
```



Для создания нескольких ссылок в одном выражении, нужно указывать знак амперсанда у каждой переменной. Кроме того, ссылка может быть константной: можно получать значение переменной, на которую ссылаемся, но нельзя изменить:

```
1 int i1 = 10, i2 = 12, i3 = 14;
2
3 int &r1 = i1, &r2 = i2;
4 const int &r3 = i3;
5 // r3 *= 3; // Не получится!
6
7 printf("i1 + i2 + i3 = %d\n", r1 + r2 + r3);
```

Плюс пример демонстрирует, что ссылка может участвовать в любом выражении. Будет использовано значение переменной, на которую и происходит ссылка.

В противоположность указателям, C++ «прячет» адрес ссылки от прямого доступа: операция взятия адреса для переменной-ссылки всегда будет возвращать адрес исходного объекта:

```
1 int i4 = 125, i5 = 555;
2 int &ref4 = i4;
3
4 printf("addr of i4    = %p\n", &i4);
5 printf("addr of ref4 = %p\n", &ref4);
6
7 ref4 = i5;
8 printf("addr of i5    = %p\n", &i5);
9 printf("addr of ref4 = %p\n", &ref4);
```

В **седьмой строке** переменной **i4** было присвоено значение переменной **i5**.

Второй вид ссылок: **ссылки на временные объекты**. По сути, всё аналогично **ссылкам на переменные**, только в качестве объектов нужно **всегда использовать** временные значения.

```
1 //double &&ref; // Не компилируется!
2 double&& rv_ref = -555.444;
3 rv_ref += 1.33;
4 printf("value of rv_ref: %.3f, addr of rv_ref: %p\n",
5        rv_ref, &rv_ref);
6
7 const int &&rv_i_ref = 333;
8 //rv_i_ref *= 10; // Не получится!
9 printf("rv_i_ref = %d\n", rv_i_ref);
```

Также можно создавать константные ссылки на временные значения.

И существенное отличие между двумя видами ссылок: константные (неизменяемые) **ссылки на переменные** могут ссылаться на временные значения, но константные **ссылки на временные объекты** не могут ссылаться на переменные.

```
1 const int &ref1 = 777; // Всё ок
2
3 int i_val = 101;
4 //const int &&ref2 = i_val; // Не сработает
```

Данная особенность **ссылок на переменные** используется при участии их в параметрах функций.

- у ссылок нет **нулевого** значения, ссылка всегда должна быть инициализирована каким-нибудь объектом;
- для получения значения используется сама переменная-ссылка, без дополнительных операторов (в противоположность разыменованию для указателей);
- при взятии адреса ссылки возвращается адрес объекта, на который она ссылается;
- **никогда** не возвращайте из **функций** ссылки на локальные переменные/временные значения;
- ссылки играют важную роль при реализации объектно-ориентированного функционала в C++, так что познакомиться с ними заранее не бесполезно.

# Ссылки в C++: хотите знать больше?

Если есть желающие посмотреть на формализацию видов ссылок: в C++ объекты (или даже обобщение — *выражения*) делятся на различные категории. На данный момент есть пять категорий, основными являются, так называемые, **lvalue** и **rvalue**. В их определении есть нюансы, но на практике можно следовать упрощённому правилу:

- **lvalue** — это те объекты/выражения, которые могут быть **левым операндом** оператора присваивания и имеют адрес. Так вот, *ссылки на lvalue* выше обозначены как **ссылки на переменные**;
- **rvalue** — значения, которые не имеют адреса (временные значения) и используются либо в качестве **правого операнда** оператора присваивания, либо в качестве *литералов типов*. *Ссылки на rvalue* — это **ссылки на временные объекты**.


Основы категорий можно подчерпнуть тут:


[https://en.cppreference.com/w/cpp/language/value\\_category](https://en.cppreference.com/w/cpp/language/value_category)



## Дополнительный тип: **закрытые перечисления**

**Перечисления** - это пользовательский тип данных, состоящий из *ограниченного* набора констант **целого типа**. По умолчанию, *базовым типом* каждой константы является **int**. В современном C++ перечисления делятся на

 **Открытые (unscoped)** - каждая константа становится доступной глобально по имени и допускается неявное приведение значений констант к значениям целочисленных типов. Также как и в языке C, ключевое слово для объявления:  
**enum**

 **Закрытые (scoped)** - каждая константа доступна только через название перечисления с использованием оператора :: и своего имени. Допускаются только **явные** преобразования в значения целочисленных типов. Ключевое слово для объявления:  
**enum class**



Синтаксис определения **закрытого перечисления**:

```
enum class <название> [: <целочисленный тип>]  
{  
    <константа_1> [= <значение_1>],  
    [<константа_2>, <константа_3>, ...]  
};
```

При определении все значения констант задаются по тем же правилам, как и для *открытых* перечислений в языке C (по умолчанию значения начинаются с нуля, увеличиваются на единицу от предыдущей константы).

# C++: закрытые перечисления

Для демонстрации отличий, что нельзя делать с переменной закрытого перечисления:

```
1 enum class Output
2 {
3     CONSOLE_TEXT, FILE_TEXT = 20,
4     FILE_BINARY,  FILE_HTML, FILE_XML
5 };
6
7 Output choice;
8 // нет названия перечисления
9 choice = FILE_XML;
10 // используется значение типа int
11 choice = 20;
12
13 // нет неявного приведения к int
14 int some_num = choice + 2;
15 bool equals_to_zero = (choice == 0);
```

Строки 9, 11, 14 и 15 вызовут ошибки компиляции программы.

# C++: закрытые перечисления

Допустимые операции:

```
1 enum class Output
2 {
3     CONSOLE_TEXT, FILE_TEXT = 20,
4     FILE_BINARY, FILE_HTML, FILE_XML
5 };
6
7 Output choice = Output::CONSOLE_TEXT;
8 // всё ок, присваиваем другую константу
9 choice = Output::FILE_TEXT;
10 // явное приведение к int
11 int status = int(choice) * 2;
12 printf("value of choice is %d\n", int(choice));
13
14 printf("Enter output source: ");
15 scanf("%d", &output);
16 printf("value of choice is %d", choice);
```

Строки **15-16** — единственные операции, когда не требуется приведение типов в явном виде. На 16 строку компилятор, возможно, выдаст только предупреждение.

# C++: закрытые перечисления

С помощью *явного приведения типов* компилятор можно обмануть:

```
1 Output choice2 = Output(777);  
2 printf("output value is %d\n", choice2);
```

Даже предупреждения не будет.

И пример явного указания типа для хранения значений переменных перечисления: сохранение набора математических операций в виде символьных констант:

```
1 enum MathOperator : char  
2 {  
3     PLUS      = '+', MINUS  = '-',  
4     MULTIPLY  = '*', DIVIDE = '/',  
5 };
```

Код работает из-за того, что тип **char** совместим с целочисленными типами.

# Возможности стандартной библиотеки: **<algorithm>**

## <algorithm>: сортировка

В C++ в библиотеке <algorithm> определена функция **std::sort**, с помощью которой можно сортировать массивы различных типов. Её сигнатура следующая (в применении к статическим массивам):

```
void std::sort(first, last,  
               comparator = operator<);
```

- 1-ый аргумент **first** - указатель (или аналог) на первый элемента сортируемого объекта
- 2-ой аргумент **last** - указатель (или аналог) на элемент, следующего за последним сортируемым
- 3-ий аргумент **comparator** - функция, которая умеет сравнивать **два** элемента сортируемого объекта. По умолчанию используется оператор «<»

Фактически, сортируется диапазон [**first**, **last**).

## <algorithm>: сортировка

Функция сравнения **comparator** должна быть определена как

```
bool comparator(Type elem1, Type elem2);
```

Функция должна возвращать

- **true**, если элемент **elem1** должен идти раньше **elem2** в упорядоченной последовательности
- **false** - иначе

Type - тип сортируемых элементов.

## <algorithm>: сортировка

Пример: сортировка действительного массива

```
1 #include <algorithm>
2
3 double my_arr[] = {55.4, 1.34, -0.95, 9.98,
4                   43.56, 3.4};
5
6 std::sort(my_arr, my_arr + 6);
7
8 printf("Ascent order:\n");
9 for (double elem : my_arr) {
10     printf("%.2f ", elem);
11 }
12 printf("\n");
```



## <algorithm>: сортировка

Пример: сортировка действительного массива по убыванию

```
1 #include <algorithm>
2
3 bool my_compr(double v1, double v2)
4 {
5     return v1 > v2;
6 }
7
8 double my_arr[] = {55.4, 1.34, -0.95, 9.98,
9                    43.56, 3.4};
10
11 std::sort(my_arr, my_arr + 6, my_compr);
12
13 printf("Sort in desc order:\n");
14 for (double elem : my_arr) {
15     printf("%.2f", elem);
16 }
17 printf("\n");
```

## <algorithm>: сортировка

Пример: сортировка части массива

```
1 #include <algorithm>
2
3 int ints[] = {3, -4, 11, 67, -2, -1
4               43, 5, 12, -9, 11, -15};
5
6 std::sort(ints, ints + 7);
7
8 printf("First seven elems in order:\n");
9 for (int elem : ints) {
10     printf("%d ", elem);
11 }
12 printf("\n");
```

## <algorithm>: обмен значениями переменных

<algorithm> предоставляет функцию **swap** для обмена значениями у двух переменных одинакового типа. Её сигнатура:  
`void swap(<тип> first, <тип> second);`

- 1-ый аргумент **first** - первая переменная
- 2-ой аргумент **second** - вторая переменная

```
1 #include <algorithm>
2
3 // Как обменивать значения в лоб
4 double var1 = 10.5, var2 = 45.6;
5 double tmp = var1;
6 var1 = var2;
7 var2 = tmp;
8
9 // а так - с помощью стандартной функции
10 std::swap(var1, var2);
```

## <algorithm>: выбор наибольшего/наименьшего

<algorithm> предоставляет функции **max** и **min** для выбора, соответственно, максимального и минимального **из двух значений**. Сигнатура:

<тип> max(<тип> first, <тип> second);

<тип> min(<тип> first, <тип> second);

```
1 #include <algorithm>
2
3 double var1 = 10.5, var2 = 45.6;
4 double max_val = std::max(var1, var2),
5        min_val = std::min(var1, var2);
6
7 printf("Max of var1 and var2 is %.3f\n"
8        "Min of var1 and var2 is %.3f\n",
9        max_val, min_val);
```