III

7 марта 2020

Символьный тип

- **char** - ещё один фундаментальный тип данных в языке C
- Используется для хранения текстовых символов. Сами символы указываются в **одинарных** кавычках, например: 'a', 'b', 'd', '4', '%', '!'
- Символы хранятся в виде **целых чисел**. Фактически, **char** является ещё одним целочисленным типом
- Это единственный тип, размер которого ограничен стандратом языка. Размер **char** **всегда** равен 1 байту
- Но всё просто не бывает: стандарт C++ не обговаривает, должен ли тип **char** быть знаковым или беззнаковым
- Существуют специальные символы, которые в текстовом виде представлены более чем одним знаком: '\n', '\t', '\r', но по сути являются одиночными
- Также стандартом определены беззнаковый **unsigned char** и знаковый **signed char** типы

Символы и кодировка

Кодировка - специальная таблица, связывающая каждый символ с соответствующим ему целым числом
Базовая для ЭВМ - **ASCII** (American standard code for interchange information, стандартная американская кодировка для обмена информацией). Основные характеристики:

- 255 символов, представленных положительными целыми кодами: от 0 до 255 (ссылка)
- В C++ в диапазон типа **char** гарантировано входят первые 128 символов
- каждый символ занимает один байт
- Включает в себя все цифры, буквы английского алфавита в нижнем/верхнем регистрах и некоторые другие символы (тильда, процент, '@', решётка и т.п.)
- Коды букв (отдельно группы в верхнем и нижнем регистрах) и цифр - идут последовательно

Символьный тип

Базовые операции с переменными типа **char**

```
1  char symbol = '%';
2  printf("%c\n", symbol); // печатаем знак процента
3
4  symbol = '#'; // знак переноса строки
5  printf("symbol is %c\n", symbol);
6
7  // А поскольку char — целочисленный:
8  symbol = '5';
9  // в арифм-х операциях участвует код символа!
10 symbol += 2;
11
12 // Выводим на экран семёрку
13 printf("%c\n", symbol);
```

Символьный тип

Можно и код из кодировки узнать, используя явное приведение типов

```
1  char sym = '9';
2
3  printf("symbol: %c\n", sym);
4  printf("Code of symbol: %d\n", int(sym));
5
6  // Использование в операторах сравнения
7  bool is_less = '2' < sym;
8  // Переменная is_less здесь равна true
9
10 // Печать английского алфавита
11 sym = 'a'
12 while (sym <= 'z') {
13   printf("%c ", sym);
14   ++sym;
15 }
16 printf("\n");
```

Символьный тип

С помощью типа **char** можно организовывать простой интерактив в программах для текстовых терминалов. Базовый шаблон:

```
1  char option;
2  printf("Want to continue (y/n)? ");
3  scanf("%c", &option);
4
5  // Проверку делаем независимо от регистра
6  if ( option == 'y' || option == 'Y' ) {
7    printf("We have the agreement\n");
8    // Полезный код появляется тут...
9  } else {
10   printf("No agreement for us...\n");
11   // Обрабатываем и отказ...
12 }
```

специальный тип данных:

Статические массивы в C++

Статические массивы в C++. Основы

Массив (в C++) - контейнер для данных, объединяющий конечное типизированное число элементов, каждому из которых сопоставляется свой **целочисленный индекс**.
Общая форма определения массива в C++:

```
<тип_данных> <имя_массива>[<размер>];
```

Задаётся тип, название массива и его размер. **Размер** массива должен быть константой.

Статические массивы в C++. Основы

Индексация элементов в массиве **начинается с нуля**. Этим **индекс** отличается от **номера элемента** массива. Оператором индексации является пара квадратных скобок **[]**.

```
1  int vec[10];
2
3  // Задаём значение первого элемента
4  vec[0] = 4;
5  // А тут — четвёртого
6  vec[3] = 55;
7
8  printf("1-st item: %d\n", vec[0]);
9  printf("4-th item: %d\n", vec[3]);
10 printf("Summ of 1-st and 4-th: %d\n",
11         vec[0] + vec[3]);
```

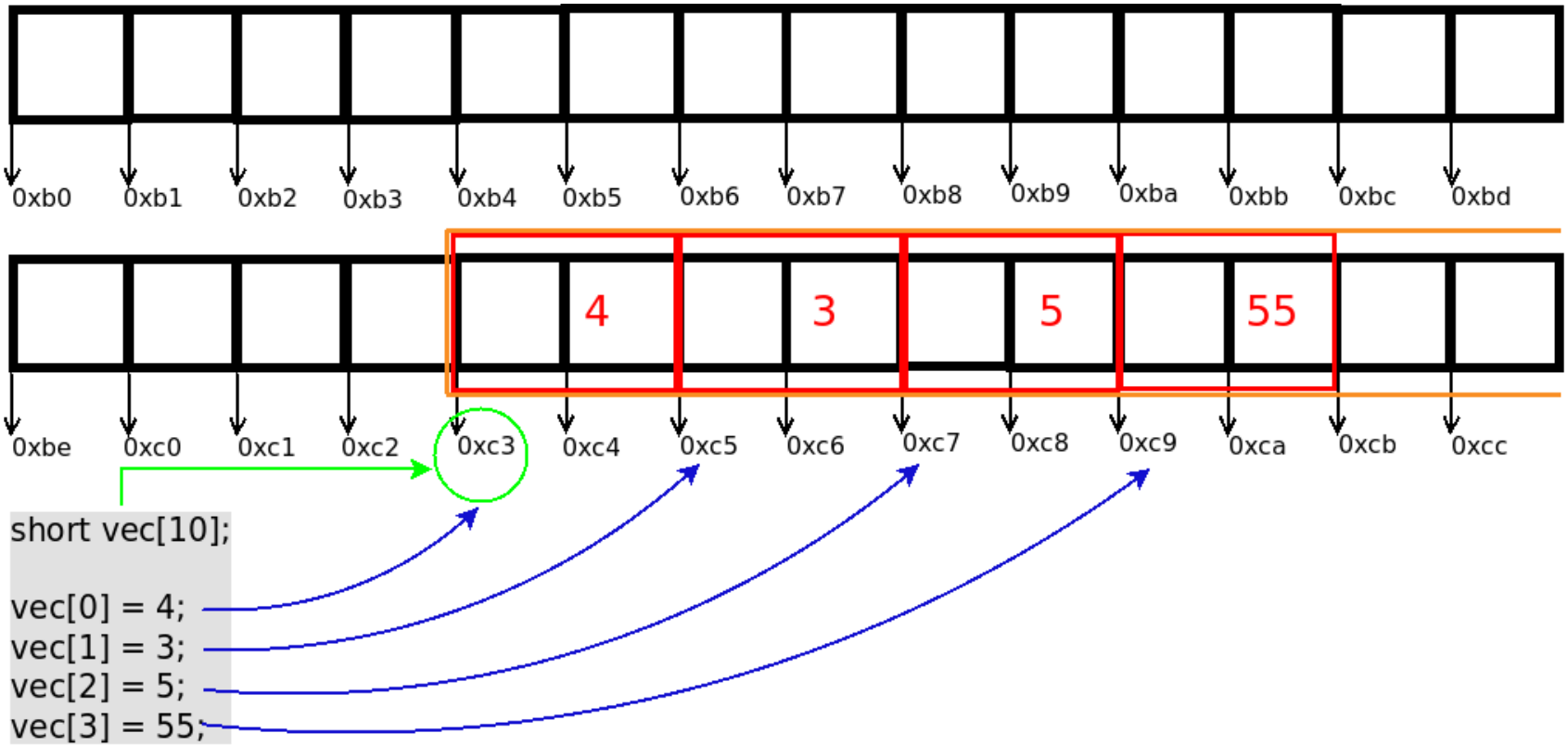
Как только был получен доступ к конкретному элементу массива с помощью индекса, с ним становятся возможны **все** операции, что определены для соответствующего типа данных. Так, в примере выше с элементами **vec[0]** и **vec[3]** работаем как с отдельными переменными типа **int**.

Замечание об индексации

Язык C++ не определяет, что должно происходить при применении индекса, который *превышает* размер массива. Такая ситуация называется **неопределённым поведением**. Компиляторы также ограничены в возможности проверить корректность индексов для массивов, используемых в программах.
Таким образом, за правильностью индексов необходимо следить самостоятельно при написании программы.

Статические массивы в C++. Основы

Точка зрения памяти. В C++ все элементы одного массива располагаются в памяти последовательно.



Адрес первого элемента является **адресом всего массива** (на картинке - адресом переменной под именем **vec**)

На предудущем слайде использован тип **short** для хранения целых чисел со знаком вместо типа **int** только из-за того, что под него, как правило, выделяется 2 байта для каждой переменной. И их компактнее размещать на поясняющей картинке.

Снова об индексации

Предыдущая картинка раскрывает суть индексации в C++. Сама переменная **vec** (переменная массива) хранит только **адрес массива** (адрес его первого элемента). А индекс означает **смещение** относительно этого адреса. Таким образом, **vec[0]** означает: взять адрес массива, пропустить от него *направо* **нуль** блоков памяти (каждый блок - равен размеру типа **short**) и считать следующий блок как значение указанного типа. Для **vec[2]** - пропускаем два блока, и считываем третий, в котором, согласно примеру, хранится число **5**.

Статические массивы в C++. Основы

Из того, что сама переменная массива является по сути его **адресом** (без применения оператора индексации), следует следующее ограничение по работе со статистическими массивами: **невозможно применить оператор присваивания** для переменных массивов, даже если их размер совпадает. На примере:

```
1 int vec[8], vec2[8], vec3[18];
2 vec[0] = 10;
3 vec[1] = 2;
4
5 // Строки ниже не пропустит компилятор
6 vec2 = vec;
7 vec3 = vec;
```

Единственный вариант для копирования одного массива в другой - использование циклов и индексов элементов.
Это первый пример, почему массивы были названы «специальным» типом данных.

Аналогично переменным, элементам массива можно присваивать начальные значения в момент его[массива] создания (**инициализация**). Для массивов инициализация выполняется с помощью пары **фигурных** скобок **{}**.

```
1 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

Создаётся массив на 8 элементов **int**, в каждый элемент сохраняется конкретное число.

```
2 double real_arr[5] = { 3.4, 5.5, 77.11 };
```

Создаётся массив на 5 элементов **double**, но значения предоставлены только для первых трёх. В этом случае, к оставшимся элементам будет применена *инициализация по умолчанию* (нулевая инициализация для фундаментальных типов) (**0** - для целых чисел, **0.0** - для действительных).

```
3 int another_vec[] = { 1, 2, 3, 4 };
```

При явной инициализации (когда количество сохраняемых значений соответствует задуманному размеру) размер можно не указывать. В примере, размер массива равен **4**.

```
4 int estimates[7];
```

Когда инициализация не выполняется, заполнение массива значениями происходит по правилам, аналогичным фундаментальным переменным. Так, если массив имеет локальную область видимости - то значения каждого из его элементов не определены. Если массив имеет глобальную область видимости - происходит заполнение *нулевыми* значениями.

Начиная со стандарта **C++11** существует специальная форма цикла **for** для перебора всех элементов массива (так называемый, *for-range* или диапазонный цикл **for**).
Его общая форма:

```
for (<имя_переменной> : <имя_массива>) {
  // работа с переменной
}
```

Работает следующим образом: **каждый** элемент массива, начиная с первого, **копируется** в переменную и выполняется итерация цикла. Типы переменной и массива **должны совпадать**.

Статические массивы в C++. Основы

На примере, печать элементов массива.

```
1 const size_t SZ = 4;
2 double rates[SZ] = { 1.1, 2.2, 5.2, 6.5 };
3
4 for (double r : rates) {
5   printf("%.5f ", r);
6 }
7 printf("\n");
```

Статические массивы в C++. Основы

На примере, печать элементов массива.

```
1 const size_t SZ = 4;
2 double rates[SZ] = { 1.1, 2.2, 5.2, 6.5 };
3
4 for (double r : rates) {
5   printf("%.5f ", r);
6 }
7 printf("\n");
```

В сравнении, «классическим» **for** печать будет выглядеть следующим образом:

```
1 const size_t SZ = 4;
2 double rates[SZ] = { 1.1, 2.2, 5.2, 6.5 };
3 // Явно используем индекс для доступа к элементу
4 for (size_t i = 0; i < SZ; i++) {
5   printf("%.5f ", rates[i]);
6 }
7 printf("\n");
```

Статические массивы в C++. Основы

Если задача состоит в изменении **всех** элементов массива последовательно и по одному сценарию, то можно ссылочной формой цикла *for-range* (вместо переменной будет использоваться ссылка на элемент массива, следовательно, отсутствует копирование).

```
1  double rates[] = { 1.1, 2.2, 3.3, 6.555 };
2  // Возводим каждый элемент массива в куб
3  for (double& r : rates) {
4    r *= r * r;
5  }
6  // Проверяем, что массив действительно поменялся
7  for (double r : rates) {
8    printf("%.4f ", r);
9  }
10 printf("\n");
```

Ссылочная форма объявляется аналогично *передаче параметров по ссылке* в функцию, с помощью знака амперсанда **&** (строка 3)

Если задача состоит в том, чтобы избежать копирования при переборе элементов массива циклом *for-range* - можно использовать **константную ссылку**.

```
1 double rates[] = { 1.1, 2.2, 3.3, 6.5, 0.3, ↩
      0.567, 0.222 };
2
3 for (const double& r : rates) {
4   printf("%.4f ", r);
5   // Так сделать компилятор не позволит:
6   // r *= 2;
7 }
8 printf("\n");
```

Ограничения цикла *for-range*

Данный вариант цикла **for** работает только тогда, когда компилятор может определить размер массива во время анализа исходного кода.

Многомерные массивы можно создавать с помощью последовательного использования пар квадратных скобок.

```
1  int matrix1[10][10];
2
3  for (int i = 0; i < 10; ++i) {
4    for (int j = 0; j < 10; ++j) {
5      matrix1[i][j] = i + j;
6    }
7  }
8  // Так хитро можно напечатать двухмерный массив
9  for (auto& row : matrix1) {
10   for (int elem : row) { printf("%d ", elem); }
11   printf("\n");
12 }
13 printf("\n");
```

Как правило считают, что первый индекс отвечает за строки, второй - за столбцы.

Многомерные массивы также можно *инициализировать* на этапе создания. Синтаксис демонстрируется следующим примером

```
1 // Инициализация
2 int matrix2[3][3] = { {1, 2, 3},
3                       {4, 5, 6},
4                       {7, 8, 9} };
```

Создаём массив **3x3**, каждой строке присваиваем по тройке чисел. Каждая пара фигурных скобок внутри общих соотвествует одной строке.

Многомерные массивы: печать двумерного массива может быть выполнена с помощью индексов

```
1  // Размер лучше делать глобальной константой
2  const size_t SZ = 10;
3
4  // Создаём матрицу и как-нибудь её заполняем
5  int matrix[SZ][SZ] = { {...}, ... };
6
7  print("Заданная матрица:\n");
8  for (size_t i = 0; i < SZ; ++i) {
9    for (size_t j = 0; j < SZ; ++j) {
10     printf("%d ", matrix[i][j]);
11   }
12   printf("\n");
13 }
14 printf("\n");
```

Пример: перемножение целочисленных матриц

```
1  const size_t SZ = 4;
2  int matrix1[SZ][SZ], matrix2[SZ][SZ];
3  for (size_t i = 0; i < SZ; i++) {
4    for (size_t j = 0; j < SZ; j++) {
5        matrix[i][j] = (i + 1) * (j + 1) ;
6        matrix[i][j] = i + j + 1;
7    }
8  }
9
10 int matrix_prod[SZ][SZ] = { {0}, {0}, {0}, {0} };
11 for (size_t i = 0; i < SZ; i++) {
12   for (size_t j = 0; j < SZ; j++) {
13     matrix_prod[i][j] += matrix1[i][j] * matrix2[↩
         j][i];
14   }
15 }
16
17 // Вывести matrix_prod на консоль
```

Многомерные массивы: в числе размерностей никто не ограничен. Гарантируется работа до 31 уровня вложенности. Как пример, массив шестимерных точек.

```
1 int monster_points[3][4][5][3][4][5];
2 // Задание всего лишь одной координаты
3 monster_points[0][0][0][0][0][0] = 5;
```

Для использования подобных многомерных массивов необходимо очень хорошо представлять, какой набор данных будет корректно и очевидным образом описан с их помощью.

Передача статических массивов в функции тоже работает не совсем очевидным способом. Как и с переменными, массив в функцию можно передать как **по значению**, так и **по ссылке**. Но в первом случае произойдёт не копирование массива, как можно было бы предположить по аналогии с переменными, а **копирование его адреса**. Самое неприятное следствие из такого поведения состоит в том, что при наличии параметра массива у функции, внутри её тела невозможно узнать размер переданного массива.

При передаче массива по значению в функцию:

- размерность массива можно не указывать, она не влияет ни на что с точки зрения компилятора;
- следует передавать актуальный размер массива отдельным параметром.

Статические массивы в C++. Основы

Как простой пример: печать массива целых чисел на экран.

```
1  void show_array(int arr[], size_t count)
2  {
3    for (size_t i = 0; i < count; ++i) {
4      printf("%d ", arr[i]);
5    }
6    printf("\n");
7  }
8
9
10 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 },
11     rates[16] = {0};
12 show_array(vec, 8);
13 show_array(rates, 16);
```

В функции **show_int_array** первым параметром идёт массив целых чисел, у него указана пара квадратных скобок, но не указан конкретный размер.

Никакой проверки размерности массива не происходит.

```
1  void show_array(int arr[55], size_t count)
2  {
3    for (size_t i = 0; i < count; ++i) {
4      printf("%d ", arr[i]);
5    }
6    printf("\n");
7  }
8
9
10 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
11 show_array(vec, 8);
```

При таком определении функции, всё будет работать без проблем, как и ранее. При передаче одномерного массива в функцию **по значению** - можно передавать массив любой размерности.

Передача массива по значению делает **невозможным** использование цикла *for-range*

```
1 void show_array(int arr[], size_t count)
2 {
3   // Так компилятор не пропустит
4   for (int elem : arr) {
5     printf("%d ", elem);
6   }
7   printf("\n");
8 }
```

Статические массивы в C++. Основы

Другая ситуация с многомерными массивами: нужно указать все размерности, **кроме первой**.

```
1  const size_t COLS = 8;
2
3  void show_2D_array(int matr[][COLS], size_t ↩
       rows_count)
4  {
5    for (size_t i = 0; i < rows_count; ++i) {
6      for (size_t j = 0; j < COLS; ++j) {
7        printf("%d ", matr[i][j]);
8      }
9      printf("\n");
10   }
11   printf("\n");
12 }
```

И эта функция будет принимать двухмерный массив, в котором число строк может быть произвольным, но число столбцов **должно быть равным** восьми (в данном примере).

Статические массивы в C++. Основы

Тем не менее, остаётся способ передавать в функцию статические массивы конкретного размера с помощью **передачи по ссылке**. Для этого имя параметра-массива заключается в круглые скобки и перед самим именем ставится знак **&**

```
1  const size_t COLS = 8;
2
3  void fill_eighth_array(int (&arr)[COLS])
4  {
5    for (int& elem_ref : arr) {
6      elem_ref = rand_a_b(-5.0, 5.5);
7    }
8  }
9
10 int rates[8], scores[12];
11 fill_eighth_arra(rates); // Всё ок
12 // fill_eighth_arra(scores); // Нельзя так
```

В созданную функцию можно передать только массивы, размерность которых равна восьми.

Аналогично, можно придумать функцию, которая будет работать только с действительными матрицами размерностью, скажем, 4x4

```
1  void fill_matrix4x4(double (&matrix)[4][4],
2                      double a, double b)
3  {
4    for (size_t i = 0; i < 4; i++) {
5      for (size_t j = 0; j < 4; j++) {
6        matrix[i][j] = rand_a_b(a, b);;
7      }
8    }
9  }
10
11 double matr[4][4];
12 fill_matrix4x4(matr, 0.0, 1.5);
```

Функция заполняет матрицу 4x4 случайными числами в диапазоне $[a; b]$. Двумерные массивы других размерностей компилятор в подобную функцию просто не пропустит.

Статические массивы в C++. Резюме

- Статические массивы в C++ индексируемы, индексы начинаются с **нуля**
- Массивы в C++ типизированны.
- Длина массива задаётся при его создании и должны быть константой
- Доступ к элементам происходит через оператор индексации
- Элементы массивов расположены в памяти **непрерывно**
- Имя переменной-массивы связано с его адресом
- Массивы могут быть многомерными
- Передавать статический массив в функции можно как *по значению*, так и *по ссылке*, но в обоих случаях **не происходит** копирования элементов массива
- Статический массив **не может использоваться** в качестве возвращаемого значения из функции

Пространства имён (**namespaces**)

Пространства имён

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 стандратной библиотеки, хоть из собственной, хоть из внешней) каждая программа может добавить свой набор переменных/функций/типов.

Пространства имён

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

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

Имя пространства имён - **fkn_omsu**, внутри него содержатся — одна константа и одна функция.

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", fkn_omsu::PLAYERS);
19
20 int i1 = 12, i2 = 12;
21 bool same = fkn_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 «**fkn_omsu::**».

Пространства имён

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

```
1 using namespace fkn_omsu;
```

- данное использование **using** делает **все идентификаторы** из пространства имён **fkn_omsu** доступными в текущей **области видимости** (другими словами, происходит *импорт имён*).

Возможен частичный импорт идентификаторов:

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

Пространства имён

Вся стандартная библиотека C++ заключается в пространство имён **std**. Вследствии чего, в базовом шаблоне для лабораторных работ появилась строка про запас вида:

```
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 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**, фукнция получения модуля для типов, представляющих числа с плавающей точкой, требует явной записи своего полного названия.

Пространства имён

Правило хорошего тона

В современном C++ рекомендуется по возможности использовать полный импорт идентификаторов (**using namespace fkn_omsu** и ему подобные) только внутри отдельных блоков кода, ограниченных парой *фигурных скобок* (функции, 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**).

Сортировка массивов

Сортировка массивов

Два базовых упорядочения:

- по возрастанию — **asc** (от ascending - возрастающий, восходящий)
- по убыванию — **desc** (от descending - убывающий, нисходящий)

Сортировка массивов

1. Сортировка вставками: insertation sort
   Идея: «разбиваем» массив на упорядоченную и неупорядоченную части. И последовательно элементы из второй части переставляем в первую в соответствии с порядком сортировки.

Сортировка массивов

1. Сортировка вставками: insertation sort
   Идея: «разбиваем» массив на упорядоченную и неупорядоченную части. И последовательно элементы из второй части переставляем в первую в соответствии с порядком сортировки.

```
1 for (int j = 1; j < ARR_SZ; j++) {
2   int key = arr[j];
3   int i = j - 1;
4   while (i > 0 && arr[i] > key) {
5     arr[i + 1] = arr[i];
6     i--;
7   }
8   arr[i+1] = key;
9 }
```

Сортировка массивов

2. Сортировка обменами: bubble sort
   Идея: проходим последовательно по парам соседних элементов массива и попарно меняем в нужном порядке.

Сортировка массивов

2. Сортировка обменами: bubble sort
   Идея: проходим последовательно по парам соседних элементов массива и попарно меняем в нужном порядке.

```
1 for (int step = 0; step < ARR_SZ; step++) {
2   for (int i = 0; i < ARR_SZ - 1; i++) {
3     if (arr[i] > arr[i + 1]) {
4       swap(arr[i], arr[i + 1]);
5     }
6   }
7 }
```

Сортировка массивов

3. Сортировка слиянием: merge sort
   Идея: разделяем массив на половины — каждую половину сортируем отдельно — соединяем упорядоченные половины в массив. Разделение идёт рекурсивно до тех пор, пока не получим массив размера 1 (то есть — конкретный элемент).

Сортировка массивов

3. Сортировка слиянием: merge sort
   Идея: разделяем массив на половины — каждую половину сортируем отдельно — соединяем упорядоченные половины в массив. Разделение идёт рекурсивно до тех пор, пока не получим массив размера 1 (то есть — конкретный элемент).

```
1  void merge_sort_asc(int arr[],
2                      int first, int last)
3  {
4    if (first < last) {
5      int middle = (first + last) / 2;
6
7      merge_sort_asc(arr, first, middle);
8      merge_sort_asc(arr, middle + 1, last);
9
10     direct_merge_asc(arr, first, middle, last);
11   }
12 }
```

Сортировка массивов

3. Сортировка слиянием: merge sort

3. Сортировка слиянием: merge sort

```
1  void direct_merge_asc(int arr[], int f, int m, int l)
2  {
3    int l_pos = f, r_pos = m + 1;
4    int l_lim = m + 1, r_lim = l + 1, total_len = l - f + 1;
5
6    for (int step = 0; step < total_len; step++) {
7      if (l_pos < l_lim && r_pos < r_lim) {
8        if (arr[l_pos] < arr[r_pos]) { l_pos++; } else {
9          int current = arr[r_pos]; bool updated = false;
10         for (int i = r_pos; i > l_pos; i--) {
11           arr[i] = arr[i-1]; updated = true;
12         }
13         arr[p + step] = current; r_pos++;
14
15         if (updated) {
16           l_pos++; l_lim++;
17         }
18       }
19     }
20   }
21 }
```

Стандартный способ сортировки: знакомство с **<algorithm>**

Сортировка массивов

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** - адрес первого сортируемого элемента
- 2-ой аргумент **last** - адрес элемента, следующего за последним сортируемум
- 3-ый аргумент **comparator** - функция, которая умеет сравнивать **два** элемента массива. По умолчанию используется оператор «<»

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

Сортировка массивов

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.

Сортировка массивов

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  print("Упорядочение по возрастанию:\n");
9  for (double elem : my_arr) {
10  printf("%.2f ", elem);
11 }
12 printf("\n");
```

Сортировка массивов

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");
```

Сортировка массивов

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");
```

Обмен значениями переменных

<algorithm> предоставляет функцию **swap** для обмена значениями у двух переменных одинакого 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 обменять 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);
```

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