Лекция III

5 октября 2018

специальный тип данных:

Статические массивы в C++

Статические массивы в C++. Основы

Массив (в программировании) - структура данных, объединяющее конечное число элементов под одним именем (идентификатором), каждому из которых сопоставляется свой **целочисленный индекс**.
В языке C++ массивы являются типизированными, то есть могут содержать элементы только одного типа.
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 в C++ **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  print("1-ый элемент: ", vec[0]);
9  print("4-ый элемент: ", vec[3]);
10 print("Сумма 1-го и 4-го: ", vec[0] + vec[3]);
11
12 // Поскольку второй элемент не был задан, язык
13 // не гарантирует конкретного значения, которое
14 // окажется во втором элементе массива.
15 print("2-ой элемент: ", vec[1]);
```

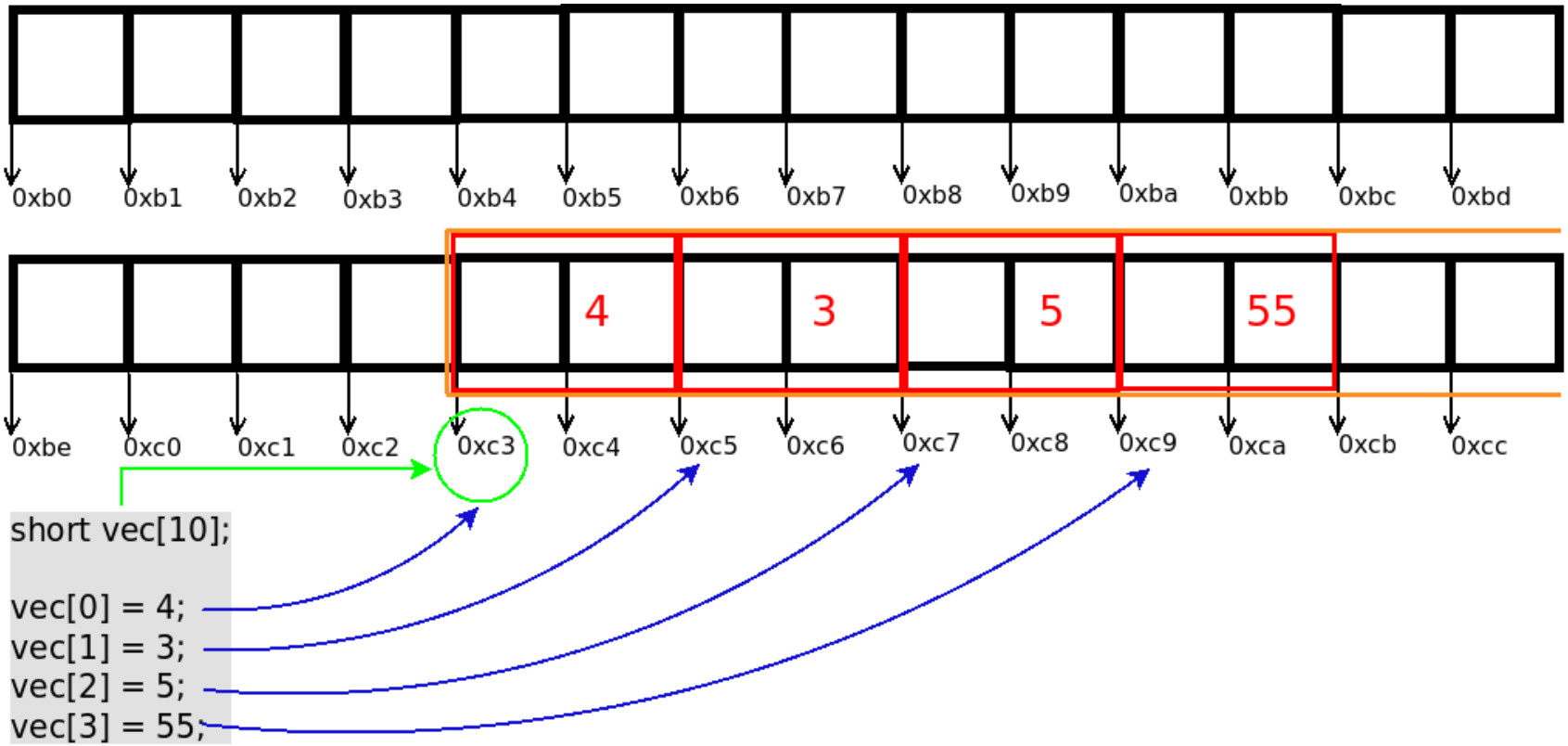
Как только был получен доступ к конкретному элементу массива с помощью индекса, с ним становятся возможны **все** операции, что определены для соответствующего типа данных. Так, в примере выше с элементами **vec[0]** и **vec[3]** работаем как с отдельными переменными типа **int**.

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.

Статические массивы в C++. Основы

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, оставшимся элементам будет присвоен **нуль** (**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**.

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

На примере, печать элементов массива.

```
1 double rates[] = { 1.1, 2.2, 5.2, 6.5 };
2
3 for (double r : rates) {
4   print(r, " ");
5 }
6 print("\n");
```

Здесь переменная **r** для записи каждого значения массива создаётся только для цикла.

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

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

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    print(r, " ");
9  }
10 print("\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 // Проверяем, что массив действительно поменялся
4 for (const double& r : rates) {
5   print(r, " ");
6   // Так сделать компилятор не позволит:
7   // r *= 2;
8 }
9 print("\n");
```

Ограничения цикла *for-range*

Данный вариант цикла **for** работает только тогда, когда объявление массива и цикл находятся в **одной** области видимости.

Статические массивы в C++. Основы

Общий шаблон для решения некоторых задач с помощью статических массивов.

```
1  // Определяем некоторую максимальную размерность
2  const size_t MAX_SZ = 45;
3
4  // Создаём массив максимальной размерности
5  int counters[MAX_SZ];
6  // Переменная под хранение актуального размера
7  size_t actual_sz;
8  get_value(actual_sz, "Введите число элементов:");
9
10 if (actual_sz <= MAX_SZ) {
11   // Работаем с массивом здесь.
12   // Заполнение значениями, вывод на экран
13   // и другие операции... Например:
14   for (size_t i = 0; i < actual_sz; i++) {
15     counters[i] = i * i;
16   }
17 }
```

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) { print(elem, " "); }
11   print("\n");
12 }
13 print("\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 числами
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     print(matrix[i][j], " ");
11   }
12   print("\n");
13 }
14 print("\n");
```

Пример: заполнение двухмерного массива «змейкой»

```
1  const size_t SZ = 10;
2  int matrix[SZ][SZ];
3  for (size_t i = 0; i < SZ; i++) {
4    for (size_t j = 0; j < SZ; j++) {
5      if (i % 2 == 0) {
6        matrix[i][j] = SZ * i + j + 1;
7      } else {
8        matrix[i][SZ - j - 1] = SZ * i + j + 1;
9      }
10   }
11 }
12 print("Заданная матрица:\n");
13 for (size_t i = 0; i < SZ; ++i) {
14   for (size_t j = 0; j < SZ; ++j) {
15     print(matrix[i][j], " ");
16   }
17   print("\n");
18 }
19 print("\n");
```

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, какие данные будут корректно и очевидным образом описаны 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_int_array(int arr[], size_t count)
2  {
3    for (size_t i = 0; i < count; ++i) {
4      print(arr[i], " ");
5    }
6    print("\n");
7  }
8
9  ...
10
11 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 },
12     rates[16] = {0};
13 show_int_array(vec, 8);
14 show_int_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. При вызове функции следует передавать корректные значения количества элементов массива.

Никакой проверки размерности массива не происходит.

```
1  void show_int_array(int arr[55], size_t count)
2  {
3    for (size_t i = 0; i < count; ++i) {
4      print(arr[i], " ");
5    }
6    print("\n");
7  }
8
9  ...
10
11 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
12 show_int_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_int_array(int arr[], size_t count)
2 {
3   // Так компилятор не пропустит
4   for (int elem : arr) {
5     print(elem, " ");
6   }
7   print("\n");
8 }
```

Статические массивы в C++. Основы

Другая ситуация с многомерные массивы: 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        print(matr[i][j], " ");
8      }
9      print("\n");
10   }
11   print("\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_incl(-50; 50);
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_incl(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
- Массивы не совсем обычный тип данных, к ним можно применять только 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

Прежде, чем идти дальше

Явное приведение фундаментальных типов

При разборе числовых типов было сказано про *неявные* преобразование целых в действительные числа и обратно. Например, при использовании значений разных типов в одном арифметическом выражении. Однако для базовых типов C++ позволяет сделать приведение *явным*.

В общей форме явные преобразования имеют вид:

```
<тип>( <имя_переменной или значение>);
```

На примере:

```
1 double real_num   = 789.354;
2 int i_num         = int(real_num);
3 double other_num = double(i_num);
```

Обратите внимание, явное преобразование в такой форме работает **исключительно** для фундаментальных типов данных.

Работа с текстом и символами в C++
Часть 1

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**. Как правило, входит в любую другую расширенную кодировку. Основные характеристики

- 128 символов
- целочисленные коды: от нуля до 127
- 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  print(symbol, "\n"); // печатаем знак процента
3
4  symbol = '#'; // знак переноса строки
5  print("Выводим символ решётки ", symbol, "\n");
6
7  // А поскольку char — целочисленный:
8  symbol = '5';
9  symbol += 2;
10
11 // Выводим на экран семёрку
12 print(symbol, "\n");
```

Символьный тип

Можно и код из кодировки узнать, используя явное приведение типов

```
1  char sym = '9';
2
3  print("Значение sym: ", sym, "\n");
4  print("Код sym: ", int(sym), "\n");
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   print(sym);
14   ++sym;
15 }
```

Символьный тип

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

```
1  char option;
2  print("Хотите продолжать (y/n)? ");
3  get_value(option);
4
5  // Проверку делаем независимо от регистра
6  if ( option == 'y' || option == 'Y' ) {
7    print("Человек согласился!");
8    // Полезный код появляется тут...
9  } else {
10   print("Согласие не получено!");
11   // Обрабатываем и отказ...
12 }
```

Символы и кодировка

Ещё одна кодировка, про которую полезно знать:
UTF8

- символ может состоять от 1 до 4 байт (следовательно, от 8 до 32 бит)
- вводится понятие «code point», равное 1 байту (8 битам) символа
- включает в себя все возможные региональные алфавиты
- Коды от 0 до 127 совпадают с кодировкой ASCII
- Получила распостранение во всех используемых языках программирования

Работа с текстом

Базовая строка в C++ - это одномерный массив значений типа **char**, последним символом которой считается специальный нулевой символ (он же - *символ окончания строки*): \0 (целочисленный код - 0). Символ конца строки **не обязан совпадать** с последним элементом массива.

```
1  char phrase1[] = "Первая строка!\n";
2  print(phrase1);
3
4  char phrase2[] = { 'A', 'B', 'C', '\0' };
5  // Тоже самое, что и:
6  // char phrase2[] = "ABC";
7  print("Второй символ в phrase2: ",
8        phrase2[1], "\n");
9
10 // А это не строка, а массив из 15
11 // элементов типа char
12 char symb_array[15];
```

Длиной строки считается количество значений типа **char**, **за исключением** символа окончания строки. Фактически, длина строки измеряется в байтах, а не текстовых символах.

Работа с текстом

Данный вид строк появился в C++ из языка C. И определение строки выше накладывают некоторые ограничения на удобство их использования.

Какие проблемы?

- строка задаётся как массив типа **char** - следовательно, пропадает возможность простого присваивания одной переменной строки другой;
- необходимо помнить про символ окончания строки \0 при переборе всех элементов массива;
- базовые строки не расширяемы: размер массива, использованного под строку, уже не изменить во время работы программы.

Конечно, часть проблем решается с использованием функций из стандартной библиотеки языка C++. Но, например, часть этих функций заставляет работать с адресами памяти отдельных ячеек массива, который содержит в себе строку.

Работа с текстом. Стандартная библиотека

Для простого примера - вычисление длины базовой строки (**в байтах**). Для этого подключается библиотека:

```
1 #include <cstring>
```

и используется функция из неё:

```
size_t strlen(str)
```

```
1  // На экране покажется число 8
2  print("Длина строки: ", strlen("A string"),"\n");
3
4  char text[] = "Строка на русском";
5  /* А здесь — зависит от ОС и её локальных
6     настроек. В русскоязычных Windows 7, 8,
7     10 — скорее всего покажет длину равную 17.
8     В ОС, основанных на Linux, наиболее
9     вероятное значение — 32 */
10 print("Длина строки: ", strlen(text), "\n");
```

Работа с текстом

Как решаются другие проблемы (поиск, копирование и т.д.) - можно посмотреть в прошлогодних лекциях по данному курсу.
Для работы с текстом в C++ гораздо легче воспользоваться стандартной библиотекой и предоставляемым типом данных - **string**.
Идея данного типа проста: хочется работать со строкой, как с некотором объектом , для которого определены ожидаемые операции. В частности, простое создание этих самых объектов, копирование, добавление текста в существующую строку, сравнение строк и тому подобное.

Работа с текстом. C++ строки

Для работы с типом **string** нужно подключить следующую библиотеку (либо использовать <**ffhelpers.h**>)

```
1 #include <string>
```

Какие преимущества даёт по сравнению с прямым использованием массивов типа **char**?

- Автоматическое выделение памяти под строку при её создании и изменении (не нужно указывать конкретный размер строки).
- Использование привычных операторов сравнения, например: $>$, $<$, $==$, $!=$
- Использование оператора $+$ для объединения строк

Уточнение

Формально, полное название типа - **std::string**, то есть он определён внутри пространства имён **std**. Далее в примерах предполагается по умолчанию, что команда **using namespace std;** добавлена в них.

Работа с текстом. C++ строки

Сами значения строк также задаются в двойных кавычках

```
1 string s1 = "Строка 1", s2;
2 s2 = "Строка 2"; // Оператор присваивани работает
3 print(s1, "\n", s2, "\n");
```

К переменным типа **string** применима операция индексации. Но по индексу можно получить только соотвествующий символ типа **char**

```
4 string s3 = "English-based string";
5 print(s3[0], "-", s3[2], "-", s3[4], "\n");
6
7 print("Строка s3 посимвольно:\n");
8 for (char sym : s3) { print(sym, " "); }
9 print("\n");
```

С помощью оператора «+» создаётся новая строка, состоящая из его операндов (значений слева и справа от оператора)

```
10 string s4 = s1 + ";" + s2;
11 print(s4, "\n");
```

Работа с текстом. C++ строки

В C++ нельзя переносить двойные кавычки на новую строку. Но для ввода многострочного текста, можно воспользоваться одной особенностью компилятора:

```
1 string text = "Весь мир насилья мы разрушим\n"
2               "До основанья, а затем\n"
3               "Мы наш, мы новый мир построим,\n"
4               "Кто был никем — тот станет всем!";
5
6 print("Какой-то куплет:\n", text, "\n");
```

Компилятор C++ объединяет все строки в двойных кавычках, идущие подряд и разделённой **только** пробелами или переносами строк, в одну непрерывную строку. Таким образом, в примере весь текст сохраняется в переменную **text**. Но при этом, символ переноса строки надо добавлять руками.

Работа с текстом. C++ строки

Получение длины (числа байт) строки.

```
size_t str.size()
size_t str.length()
```

здесь **str** - это переменная типа **string**.

```
1 string s1 = "whatever somewhere";
2
3 // Выведет 18
4 print("Длина s1: ", s1.size(), "\n");
5
6 // Также как и с базовыми строками, конкретное
7 // значение зависит от ОС
8 string s2 = "Провение технических работ";
9 print(s2.length(), "\n");
```

Это первый пример работы с переменными типа **string** с помощью специальных функций, определённых **только для них**. Такие «функции» в C++ получили название **методов**.

Работа с текстом. C++ строки

Сравнение строк: для полного сравнения строк используются привычные операторы сравнения: $>$, $>=$, $<$, $<=$, $==$, $!=$

```
1  string s1 = "France",
2         s2 = "Russia";
3
4  if ( s1 < s2 ) {
5    print("Кто бы сомневался\n");
6  }
7
8  // Проверка на равенство
9  bool is_equals = s1 == s2;
10 // is_equals равен false
```

Работа с текстом. C++ строки

Сравнение строк: как операторы сравнения $>$, $>=$, $<$, $<=$, $==$, $!=$ работают.

Строка s1 считается **меньше**, чем s2 в двух случаях (s1 $<$ s2)

1. первый несовпадающий символ в s1 имеет **целочисленный код** меньше, чем таковой в s2
2. все символы из s1 совпадают с начальными символами в s2, но вторая строка имеет **большую длину**

Строка s1 считается **больше**, чем s2 в двух случаях (s1 $>$ s2)

1. первый несовпадающий символ в s1 имеет **целочисленный код** больше, чем таковой в s2
2. все символы из s1 совпадают с начальными символами в s2, но вторая строка имеет **меньшую длину**

Иначе - строки считаются **равными**. Это так называемое, *лексикографическое* сравнение строк.