Язык программирования C++

Конспект лекций

Содержание

Лекция 1.

TODO

Лекция 2.

TODO

Лекция 3

TODO

Лекция 4.

Функции.

Программы на C++ выглядят, как множество функций, которые друг друга вызывают, поэтому фактически невозможно запустить код вне функций.
Уже знакомая нам функция — `main`. Программа запускает ее автоматически, а в конце получает от нее: успешно ли выполнилась программа.

Очевидно, создавать можно и свои функции. Рассмотрим пример:

```
int add(int a, int b) {
  return a + b;
}
```

Сначала задается тип возвращаемого результата функции, после – название, а наконец аргументы, каждый с своим типом данных. Каждая функция заканчивается ключевым словом `return`, который возвращает значение функции.
Команда `return` не обязятельно должна стоять в самом конце кода функции. Например, их можно поставить несколько, некоторые в середину

```
int add(int a, int b) {
  if (a == 0 && b == 0){
    return 0;
  }
  return a + b;
}
```

Важно помнить, что на `return` функция возвращает свое значение и заканчивает выполнение.

Есть функции, которые ничего не возвращают:

```
void foo() {
  std::cout << "nothing" << std::endl;
}
```

Вызываются функции любые функции одинаково:

```
int main() {
  std::cout << add(1, 2) << std::endl;
  std::cout << add(3, 3) << std::endl;

  foo();
  foo();
}
```

На самом деле, функция `main` тоже принимает аргументы...
TODO

Лекция 5.

TODO

Лекция 6.

Строки.

Немного общее про хранение символов. Поскольку символов много, а их все надо как-то однозначно определить, придумали кодировки. Они позволяют хранить в памяти просто номер код символа, а программа уже сама будет по этому коду определять, какой символ рисовать.
Например, есть кодировка `utf-8`. Она хранит символы Unicode и занимает 8 бит или 1 байт. Коды символов имеют полную обратную совместимость с 7-битной кодировкой ASCII.
В C++ по умолчанию используется `utf-8`.

В C++ строку можно хранить, например, как просто массив элементов типа `char`. Будет не более 256 различных значений.

```
char str[] = {'H', 'e', 'l', 'l', 'o'};
std::cout << str << std::endl;
```

На этот код санитайзер будет ругаться. Для вывода этого массива в `std::cout` передается указатель на первый символ, `char* ptr = &str[0]`. Выходит так, чтобы вывести строку, программа должна пройтись по памяти, начиная с адреса начала строки, однако она не знает, где остановиться, и выходит за его пределы. Но далее программа может просто дойти до конца выделенной ей памяти и сломаться — что она и делает. В этот момент санитайзер предупреждает, что мы вышли за пределы памяти.

Тогда можно выводить строку такой функцией:

```
void print(const char* str, size_t size) {
  while (size > 0) {
    std::cout << *str << std::flush;
    ++str;
    --size;
  }
}
```

Или можно задавать размер заранее, прямо в строке:

```
// const char str[] = {0, 0, 0, 5, 'H', 'e', 'l', 'l', 'o'}; //первые 4 байта -- размер
строки

void print(const char *str) {
  size_t size = 0;
  for (int i = 0; i < 4; ++i) {
    size <<= std::numeric_limits<unsigned char>::digits;
    size |= *str++;
  }

  while (size > 0) {
    std::cout << *str << std::flush;
    str++;
    size--;
  }
}
```

Теперь о строках. В каждой строке, помимо самих символов стоки, есть и специальный символ конца строки: `'\o'`. Поэтому, опеределив строку через двойные кавычки, мы можем выводить так:

```
void print0(const char* str) {
  while (*str != '\0') {
    std::cout << *str++;
  }
}

int main() {
  char str[] = "abcde";
  print0(str);

  return 0;
}
```

Однако, и у строк есть свои минусы: мы не можем знать длину строки за O(1). Чтобы узнать длину, мы должны пройти по всей строки, от адреса начала, до символа `'\o'`.

Структуры.

Хорошо бы уметь хранить строку и ее размер в одном месте, в одной структуре. Для этого есть структуры.

```
struct StringRef {
  const char* begin;
  size_t size;
}

struct OwningString {
  char str[100];  // здесь 100 -- длина строки, следовательно займется память на 100 чаров
  // и тут еще будет char padding[7]. (выравнивание)
  size_t size;
}
```

Тогда остальные функции мы можем переписать так:

```
void print(StringRef str) {
  for (size_t i = 0; i < str.size; ++i) {
    std::cout << str.begin[i] << std::endl;
  }
}

int main() {
  const char* ptr = "abcdefg";  // где-то в памяти лежит это строка.
  StringRef str{ptr, 7};
  OwningString str2{"ABCD", 4};
}
```

Инициализация структуры.

Можно просто сначала инициализировать и потом поименно прописывать каждое поле:

```
int main() {
  StringRef str;
  str.size = 3;
  beging = "123"
}
```

Можно сразу прописывать при инициализации, но тут важно соблюдать порядок, как они расположены в определении, и заполнять все значения:

```
int main() {
  StringRef str{"123", 3};
  // StringRef str = {"123", 3}; // Можно еще так
}
```

Начиная с C++20 появилась новая схема:

```
int main () {
  StringRef str{
    .begin = "123",
    .size = 3,
  } // Тут надо писать их в том порядке, в каком они задекларированы в определении структуры
}
```

Обращаться к полям структуры.
Самое простое – через точку:

```
std::cout << str.size << std::endl;
```

Если у нас есть указатель на структуры:

```
StringRef ref;
StringRef* ptr = &ref;

(*ptr).size = 42;
// или
ptr->size = 13;
// или
const auto& [begin, size] = ref; // называется распаковка.
// Вся структура разбивается на переменные.
// В квадратных скобках пишутся имена, по которым можно найти переменную.
```

Slightly off topic.

Шрифты.

```
snake_case,
CamelCase,
kebab-case.
```