Лекция 2. Переменные, ссылки, массивы

Илья Макаров

ИТМО JB

14 сентября 2021

Санкт-Петербург

Еще раз о переменных

- Имя переменной не должно начинаться с цифры.
- Имя переменной не должно включать следующие символы:
  1. , /, :,
  2. *, ?, .,
  3. <, >, |,
  4. …
- Имена, начинающиеся на `__`, зарезирвированны стандартом. Например, `__func__`

Как реализовать swap

Рассмотрим функцию, меняющую параметры местами:

```
void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int k = 10, m = 20;
    swap(k, m);
    std::cout << k << ' ' << m << std::endl; // 10 20
    return 0;
}
```

Проблема: swap изменяет локальные копии переменных k и m.

Ссылки

- **Ссылка** — это некоторый “синоним” для имени переменной.
- Ссылки бывают 2 типов: **константная** и **неконстантная**. Например:

```
int i = 3; // переменная типа int
int & i_ref = i; // ссылка на i
const int & i_cref = i; // константная ссылка на i
```

- Аналогичные записи с некоторым “синтаксическим сахаром”:

```
auto & i_ref = i;
const auto & i_cref = i;
int & const i_cref2 = i;
auto & const i_cref2 = i;
```

Замечание: Не путайте ссылки и указатели.

Передача параметров по ссылке

Передадим наши параметры по ссылке:

```
void swap(int & a, int & b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int k = 10, m = 20;
    swap(k, m);
    std::cout << k << ' ' << m << std::endl; // 20 10
    return 0;
}
```

Теперь swap меняет местами переменные k и m.

Замечания

- Пишите реализацию элементов из стандартной библиотеки только в образовательных целях.
- Если что-то есть в стандартной библиотеке — используйте.
- Скорее всего реализация стандартной библиотеки эффективнее.
- Например, в стандартный алгоритм `std::swap` реализован в заголовочном файле `<utility>`.
- Всегда есть но…

Указатели

- Указатель — это переменная, хранящая адрес некоторой ячейки памяти.
- Указатели являются типизированными.

```
int i = 3;
// указатель на переменную типа int
int * p = nullptr;
```

- Нулевому указателю (которому присвоено значение `0` или `nullptr`) не соответствует никакая ячейка памяти.
- Оператор взятия адреса переменной `&`.
- Оператор разыменования `*`.

```
p  = &i; // указатель p указывает на переменную i
*p = 10; // изменяется ячейка по адресу p, т.е. i
```

Еще раз о времени жизни переменных

- Переменные по времени жизни можно разделить на 2 типа: **локальные** и **глобальные**.
- Глобальные существуют все время исполнения программы.
- Локальные существуют в рамках определенной области (scope).
- Символы { и } соотвествуют началу и концу области видимости.
- В реальности все чуть сложнее…

Время жизни переменной

Следует следить за временем жизни переменных.

```
int * foo() {
    int a = 10;
    return &a;
}

int & bar() {
    int b = 20;
    return b;
}
```

```
int * p = foo(); // bad pointer
int & l = bar(); // bad reference
```

Массивы

- **Массив** — это набор однотипных элементов, расположенных в памяти друг за другом, доступ к которым осуществляется по индексу.
- Массиву, как правило, соответствует непрерывный участок памяти.
- Непрерывность участка дает преимущество — мы более эффективно используем кеш процессора.

Статические массивы

- C++ позволяет определять массивы на стеке.
- В стиле C:

```
// массив 1 2 3 4 5 0 0 0 0 0
int m[10] = {1, 2, 3, 4, 5};
```

- Аналог в стиле C++:

```
#include <array>

std::array<int, 10> m = {1, 2, 3, 4, 5};
```

Статические массивы

- Индексация массива начинается с `0`, последний элемент массива длины `n` имеет индекс `n - 1`.

```
std::array<int, 10> m = {1, 2, 3, 4, 5};

for (int i = 0; i < m.size(); ++i) {
    std::cout << m[i] << ' ';
}
```

- Или range based for:

```
for (int x : m) {
    // x - ссылка на элемент из m
    std::cout << x << ' ';
}
```

Статические массивы

- C++ умеет выводить типы самостоятельно. Например:

```
for (const auto & i : m) {
    std::cout << i << ' ';
}
```

- Важно, что ссылка константная. Ведь мы не собираемся изменять содержимое нашего контейнера.

Динамические массивы

- C++ позволяет определять массивы, размер которых не известен на момент компиляции.
- В стиле C++:

```
#include <vector>

std::vector<int> m = {1, 2, 3, 4, 5};
```

- В стиле C обсудим подробнее позже.

Динамические массивы

- Аналогичным образом можем напечать наш динамический массив:

```
for (const auto & i : m) {
    std::cout << i << ' ';
}
```

- Или так:

```
for (auto i = 0; i < m.size(); ++i) {
    std::cout << m[i] << ' ';
}
```

Почему следует использовать C++ массивы

- Обеспечивают большую безопасность.
- Достаточно унифицированы (имеют схожие интерфейсы).
- “Дружат” с алгоритмами стандартной библиотеки.
- Решают большинство ваших задач (не требуется изобретать свой велосипед).

Немного об STL

- STL = Standard Template Library.
- STL является частью стандартной библиотеки C++, описана в стандарте, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли.
- Основана на разработках для языка Ада.

Замечание: большинство подходов к разработке библиотек общего назначения и методик обобщенного программирования в целом, используемых в C++, детально описаны в книге: А. Степанов “Начала программирования”.

Составляющие STL

Можно выделить следующие основные составляющие:

- Контейнеры — способ хранения объектов в памяти.
- Итераторы — унифицированный доступ к элементам контейнера.
- Адаптеры — обёртки над контейнерами для более удобного использования.
- Алгоритмы — работа с содержимым контейнеров.
- Функциональные объекты, функторы (обобщение функций).

Чего нет в стандартной библиотеке

- Сложных структур данных.
- Сложных алгоритмов.
- Работы с графикой/звуком.
- …

Лямбда-выражения

```
auto op = [](int x, int y) { return x / y; };

// C++14
auto op = [](auto x, auto y) { return x / y; };

// то же, но с указанием типа возвращаемого значения
auto op = [](int x, int y) -> int { return x / y; };
```

Лямбда-выражения

Можно захватывать *локальные* переменные.

```
int total = 0;

// захват по ссылке
auto f1 = [&total](int x) { total += x; };

// захват по значению
auto f2 = [total](int & x) { x -= total ; };

// все по ссылке
auto f3 = [&] { return total; }

// все по значению
auto f3 = [=] { return total; }
```

Операции с массивами

Сегодня рассмотрим только 2 контейнера: `std::array` и `std::vector`, остальные рассмотрим позже.
Что можем поделать с нашими массивами? К примеру:

- Скопировать.
- Отсортировать.
- Применить некоторый функтор к элементам.
- Отфильтровать (удалить элементы).
- Применить алгоритмы из стандартной библиотеки.
- …

Операции с массивами

Как нам что-то сделать с содержимым массива?

- Написать цикл.
- Или вызвать метод контейнера.
- Или вызвать функцию из стандатной библотеки.

Пример: сортировка массива

- Объявим и заполним наш массив.

```
std::vector<int> m = {5, 2, 1, 3, 4};
```

- Вариант 1. Пишем сортировку самостоятельно. Например, пузырек:

```
for (auto i = 0; i < m.size(); ++i) {
    for (auto j = 0; j < m.size() - i - 1; j++) {
        if (m[j] > m[j + 1]) {
            std::swap(m[j], m[j + 1]);
        }
    }
}
```

- Вариант 2. Используем алгоритм `std::sort`:

```
std::sort(m.begin(), m.end());
```

Пример: копирование массива

- Объявим и заполним наш массив, создадим контейнер для копии.

```
std::vector<int> m = {5, 2, 1, 3, 4};
std::vector<int> m_copy;
```

- Вариант 1. Пишем самостоятельно:

```
for (const auto & i : m) {
    m_copy.push_back(i);
}
```

- Вариант 2. Используем алгоритм `std::copy`:

```
std::copy(m.cbegin(), m.cend(), m_copy.end());
```

- Вариант 3. Используем конструктор:

```
std::vector<int> m_copy(m.cbegin(), m.cend());
```

Пример: применение функции к массиву

- Объявим и заполним наш массив-строку.

```
std::string s("hello");
```

- Вариант 1. Пишем самостоятельно:

```
for (auto & c : s) {
    c = std::toupper(c);
}
```

- Вариант 2. Используем `std::transform`:

```
const auto f = [](char c) { return std::toupper(c); }
std::transform(s.begin(), s.end(), s.begin(), f);
```

- Вариант 3. Используем другой алгоритм из STL:

```
std::for_each(s.begin(), s.end(),
              [](char c) { std::toupper(c); });
```

Пример: удаление элементов из массива

- Объявим и заполним наш массив.

```
std::vector<int> m(10);
std::iota(m.begin(), m.end(), 0);
```

- Вариант 1. Пишем самостоятельно:

```
for (auto it = m.begin(); it != m.end();) {
    it = (*it % 2 == 0)
        ? it = m.erase(it)
        : std::next(it);
}
```

Пример: удаление элементов из массива

- Вариант 2. Используем метод:

```
bool is_odd(int i) {
    return i % 2 == 1;
}

m.erase(
    std::remove_if(v.begin(), v.end(), is_odd),
    v.end());
```

Пример: удаление элементов из массива

- Вариант 3. Используем `std::erase`:

```
bool is_odd(int i) {
    return i % 2 == 1;
}

std::erase_if(m, is_odd); // since C++20
```

Пример: удаление элементов из массива

- Объявим и заполним наш массив-строку.

```
std::string s("hello");
```

- Вариант 1. Вручную:

```
std::string s_copy = "";
for (auto & c : s) {
    if (c != 'l') {
        s_copy += c;
    }
}
```

- Вариант 2. Используем метод контейнера:

```
s.erase('l');
```

- Вариант 3. Алгоритм `std::remove_copy`:

```
std::remove_copy(str.begin(), str.end(),
                 str.begin(), 'l');
```

Пример: подсчет элементов, удовлетворяющих предикату

- Объявим и заполним наш массив.

```
std::vector<int> m(10);
std::iota(m.begin(), m.end(), 0);
```

- Вариант 1. Вручную:

```
auto counter = 0u;
for (const auto c : m) {
    if (c % 2 == 0) {
        ++counter;
    }
}
```

- Вариант 2. Используем алгоритм:

```
std::count_if(m.cbegin(), m.cend(),
              [](const auto & i) { return i % 2 == 0;
```

Ranges

- Упрощают работу с алгоритмами и контейнерами.
- Появились в C++20. Заголовочный файл `<ranges>`.
- Не все алгоритмы, адаптеры, ...были стандартизованы.
- Остальная часть широкоиспользуемых вещей доступна в библиотеке RangesV3.
- Почти все из RangesV3 скорее всего попадет в C++23.

Ranges примеры

- Объявим и заполним наш массив.

```
std::vector<int> m(10);
std::iota(m.begin(), m.end(), 0);
```

- Определим наш предикат:

```
 const auto even = [](int i) { return i % 2 == 0; };
```

- Напечатаем наш массив:

```
for (auto & i : m | std::views::filter(even)) {
    std::cout << i << ' ';
}
```

- Другой вариант:

```
for (auto & i : std::views::filter(m, even)) {
    std::cout << i << ' ';
}
```

Ranges примеры

- Дополнительно определим функтор:

```
const auto square = [](int i) { return i * i; };
```

- Выполним композицию even и square:

```
for (auto & i : m | std::views::filter(even)
                  | std::views::transform(square)) {
    std::cout << i << ' ';
}
```

Ranges примеры

- Еще раз рассмотрим пример с подсчетом четных чисел:

```
const auto even = [](int i) { return i % 2 == 0; };
const auto counter = std::ranges::count_if(m, even);
```

- Копирование массива:

```
const auto m_copy = std::ranges::copy(
    std::views::filter(m, even),
    std::back_inserter(m_copy));
```