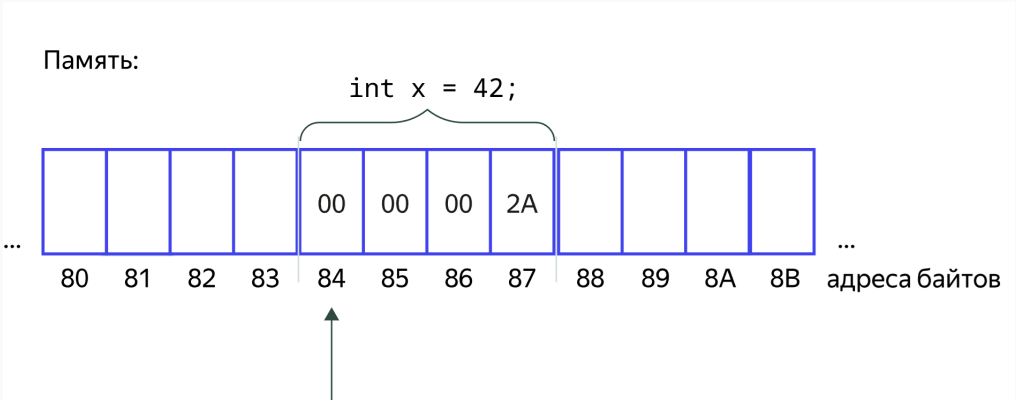
Ссылки, указатели, работа с файлами

Denis Bakin

Модель памяти и указатели

C++ — язык низкого уровня, предоставляющий **прямой доступ к памяти**.
Важно понимать, как устроена память программы и как обращаться с адресами.

- Память можно представить как **линейное пространство байт**
- **Байт** — минимальная адресуемая единица памяти
- **Битность системы** (32/64) определяет размер регистра и длину адреса



Указатели: базовое понятие

Указатель — это переменная, хранящая адрес ячейки памяти.
Компилятор “знает”, сколько байт занимает объект по этому адресу.

- `int* ptr` — указатель на `int`
- `&a` — взять адрес переменной `a`
- `*ptr` — разыменовать указатель, получить значение по адресу

```
int main() {
    int x = 42;
    int* ptr = &x;  // сохраняем адрес x в ptr

    ++x;
    std::cout << *ptr << "\n";  // 43
}
```

Адреса и порядок размещения

```
int main() {
    int x = 1;
    int y = 2;
    int z = 3;
    std::cout << &x << "\n"; // 0x7ffcdba9233c
    std::cout << &y << "\n"; // 0x7ffcdba92340
    std::cout << &z << "\n"; // 0x7ffcdba92344
}
```

- Переменные, созданные позже, часто имеют **меньший адрес**
- Разница между адресами для `int` обычно равна **4 байтам**

$$7FFCDBA92340_{16} - 7FFCDBA9233C_{16} = 40_{16} - 3C_{16} = 4_{16} = 4_{10}$$

Пример с нулевым указателем

```
int main() {
    int x = 42, y = 13;
    int* ptr = nullptr;  // нулевой указатель
    ptr = &x;
    std::cout << *ptr << "\n";  // 42
    ptr = &y;
    std::cout << *ptr << "\n";  // 13
}
```

- `nullptr` — безопасное значение для указателя
- Разыменовывать `nullptr` нельзя — приведёт к **ошибке выполнения**

Ссылки

Ссылка — это псевдоним для другой переменной. Она всегда должна быть инициализирована при создании.

```
int main() {
    int x = 42;
    int& ref = x;

    ++x;
    std::cout << ref << "\n";  // 43
    ++ref;
    std::cout << x << "\n";    // 44
}
```

- Изменения через `x` и `ref` влияют на одну и ту же область памяти
- В отличие от указателя, ссылка **не может быть переприязана**

Присвоение ссылке

```
int main() {
    int x = 42, y = 13;
    int& ref = x;
    ref = y;   // изменяет значение x, а не привязку!
    std::cout << x << '\n'; // 13
}
```

- После инициализации ссылка всегда ссылается на один и тот же объект
- Попытка “перепривязать” приведёт к **изменению исходной переменной**

Ссылки в циклах

```
std::vector<std::string> data = {"Just", "some", "random", "words"};
for (std::string &word: data) {
    std::cout << word << ' ';
}
```

- & означает, что элемент не копируется, а **берётся по ссылке**
- Копирование строк — дорогая операция
- Ссылки позволяют работать быстрее и экономнее по памяти

Висячие ссылки и указатели (dangling)

Когда объект уничтожен, а ссылка или указатель на него осталась:

```
int* ptr = nullptr;
{
    int x = 42;
    ptr = &x;
}
// здесь x уже не существует
std::cout << *ptr;  // undefined behavior
```

И аналогично со ссылкой:

```
std::vector<std::string> words = {"one", "two"};
std::string& ref = words[0];
words.clear(); // элементы удалены
std::cout << ref;  // undefined behavior
```

Работа с файлами в C++

C++ поддерживает потоки ввода/вывода файлов через `fstream`.

Пути до файлов:

- Абсолютный: `/home/user/data.txt`
- Относительный: `data/test.txt` (относительно программы)

Текстовые файлы: запись

```cpp
#include <fstream>

int main() {
    std::ofstream out("hello.txt");
    if (out.is_open()) {
        out << "Hello World!" << std::endl;
    }
    out.close();
}
```

- `std::ofstream` — поток для записи
- Если файл не существует, он будет создан
- Проверяйте `is_open()` перед записью

Текстовые файлы: чтение

```cpp
#include <fstream>
#include <string>

int main() {
    std::ifstream in("hello.txt");
    std::string line;

    if (in.is_open()) {
        while (std::getline(in, line)) {
            std::cout << line << std::endl;
        }
    }
    in.close();
}
```

- `std::getline` считывает файл **построчно**
- Удобно, если заранее неизвестно количество строк

Двоичные файлы

Иногда нужно читать и писать **сырые байты**.

```
std::vector<float> data = {0.1, 0.2, 0.3, 0.4};
std::ofstream out("data.bin", std::ios::binary);

out.write(reinterpret_cast<char*>(data.data()), data.size() * sizeof(float));
out.close();
```

И чтение:

```
std::vector<float> result(data.size());
std::ifstream in("data.bin", std::ios::binary);
in.read(reinterpret_cast<char*>(result.data()), result.size() * sizeof(float));
```

- Используем `reinterpret_cast` для приведения указателя
- Работает напрямую с байтами памяти

Пример: запись и чтение массива float

```cpp
std::vector<float> data = {1.0, 2.0, 3.0, 4.0};

std::ofstream out("foo.bin", std::ios::binary);
out.write(reinterpret_cast<char*>(data.data()), data.size() * sizeof(float));
out.close();

std::vector<float> input(data.size());
std::ifstream in("foo.bin", std::ios::binary);
in.read(reinterpret_cast<char*>(input.data()), input.size() * sizeof(float));

for (auto v : input) std::cout << v << ", ";
std::cout << std::endl;
```

- Двоичная запись быстрее и точнее, чем текстовая
- Используется, например, для хранения числовых данных и изображений

Итоги

- **Указатели** дают прямой доступ к памяти, но требуют осторожности
- **Ссылки** — безопасный способ работы с уже существующими объектами
- **Файлы** в C++ — это потоки `ifstream` / `ofstream`
- Текстовые файлы удобны для людей, двоичные — для машин
- Любая работа с памятью или файлами требует проверки ошибок