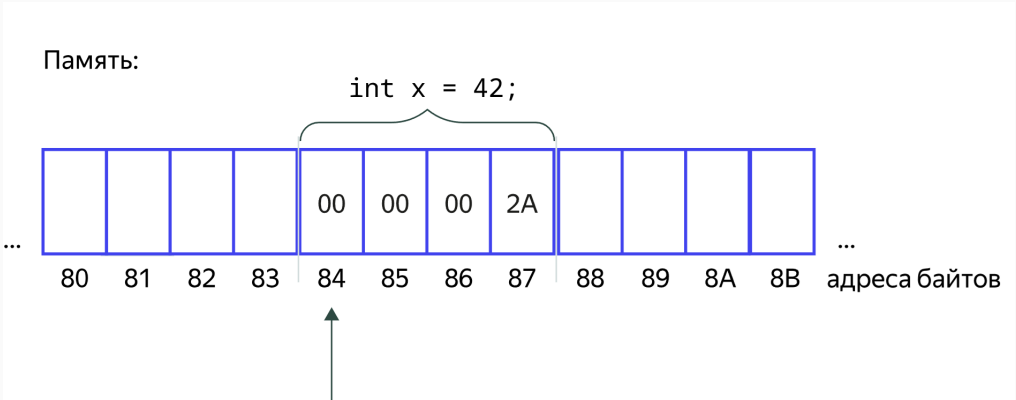
Ссылки, указатели. Стек и очередь

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
```

Стек

- Стек — структура данных с доступом только к вершине (LIFO)
- Основные операции:
  - `push(elem)` — положить элемент на вершину
  - `top()` — прочитать элемент с вершины
  - `pop()` — удалить элемент с вершины
  - `empty()` — проверка пустоты
- Интуиция: стопка тарелок, книги на столе

Стек

- Есть реализации на фиксированном и динамическом массиве
- В C++ удобно: `std::stack<T>` (ограниченный интерфейс: нет итераций/индексации)

```cpp
#include <iostream>
#include <stack>
int main() {
    std::stack<int> stack;
    stack.push(1); // push элемента на вершину стека
    // top -- получение элемента
    std::cout << stack.top() << std::endl;
    stack.pop(); // pop удаление элемента с вершины стека
    // empty -- пуст ли стек
    if (stack.empty()) {
        std::cout << "Stack is empty" << std::endl;
    } else {
        std::cout << "Stack is not empty" << std::endl;
    }
}
```

Стек: зачем?

- простая, предсказуемая структура без лишнего функционала
- часто оптимальнее, чем динамический массив для LIFO-задач
- основа исполнения вызовов функций (call stack) и рекурсии
- локальные переменные размещаются на стеке

Правильные скобочные последовательности (1 тип)

Формально:

- пустая строка — правильна
- если A и B правильны, то AB правильна
- если A правильна, то (A) (и аналогично для других скобок) правильна

Неформально:

- $(((()()()))$, $(()())()()$ – правильные
- $)($, $())()()$, $((()))$ – неправильные

Правильные скобочные последовательности (1 тип)

Правильные скобочные последовательности (1 тип)

Скобки	(	(	(	)	(	)	)	)
Баланс	_	_	_	_	_	_	_	_

Правильные скобочные последовательности (1 тип)

Скобки	(	(	(	)	(	)	)	)
Баланс	1	2	3	2	3	2	1	0

Правильные скобочные последовательности (несколько типов)

- $(([]))$, $([()])$, $[()]$ – правильные
- $([)]$, $()$, $[(])$ – неправильные

Правильные скобочные последовательности (несколько типов)

Правильные скобочные последовательности (несколько типов)

- при открывающей скобке — `push` в стек

Правильные скобочные последовательности (несколько типов)

- при открывающей скобке — `push` в стек
- при закрывающей — если стек пуст $\to$ неверно; иначе `pop` и проверить соответствие типов скобок

Правильные скобочные последовательности (несколько типов)

- при открывающей скобке — `push` в стек
- при закрывающей — если стек пуст $\to$ неверно; иначе `pop` и проверить соответствие типов скобок
- в конце стек должен быть пуст

Правильные скобочные последовательности (несколько типов)

Шаг	Ввод	Стек	Действие
1	(	(	Push ‘(’
2	[	(, [	Push ‘[’
3	{	(, [, {	Push ‘{’
4	}	(, [	Pop ‘{’
5	]	(	Pop ‘[’
6	)	пуст	Pop ‘(’
7	_	проверки	EOF

Стек минимумов

- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами

Стек минимумов

- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами
- при `push(x)` — если `x <= current_min` — `push` и в дополнительный стек

Стек минимумов

- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами
- при `push(x)` — если `x <= current_min` — `push` и в дополнительный стек
- при `pop()` — если удаляемый элемент == вершине стека минимумов — `pop` и там

Стек минимумов

- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами
- при `push(x)` — если `x <= current_min` — `push` и в дополнительный стек
- при `pop()` — если удаляемый элемент == вершине стека минимумов — `pop` и там
- `get_min()` — вершина дополнительного стека

Стек минимумов

- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами
- при `push(x)` — если `x <= current_min` — `push` и в дополнительный стек
- при `pop()` — если удаляемый элемент == вершине стека минимумов — `pop` и там
- `get_min()` — вершина дополнительного стека
- Аналогично можно реализовать стек максимумов

Стек минимумов

Операция	Основной стек	Стек минимумов	Результат
push(5)	[5]	[5]	
push(3)	[5, 3]	[5, 3]	
push(7)	[5, 3, 7]	[5, 3]	
get_min()	[5, 3, 7]	[5, 3]	3
pop()	[5, 3]	[5, 3]	
get_min()	[5, 3]	[5, 3]	3
pop()	[5]	[5]	
get_min()	[5]	[5]	5

Очередь: базовое понятие

- **Очередь (Queue)** — структура данных с принципом **FIFO**
- Аналог: очередь в кафе — обслуживаются в порядке прихода
- Основная идея — элементы выходят в том же порядке, в каком были добавлены

Очередь: базовое понятие

Основные операции:

- `push(elem)` — добавить элемент в **конец** очереди
- `tail()` — вернуть элемент с конца очереди
- `head()` — вернуть элемент с начала очереди
- `pop()` — удалить элемент с начала очереди
- `size()` — вернуть количество элементов

Очередь: применение

Очереди широко применяются при:

- обработке запросов (например, сетевые запросы, транзакции без приоритета)
- обработке событий (несинхронные производители и потребители данных)
- обходе массива **окном фиксированной длины**

Очередь: варианты реализации

Реализовать очередь можно по-разному:

- на **динамическом массиве** — просто, но неэффективно
- **циклическая очередь** на массиве фиксированной длины
- на **двух стеках** — позволяет добавить дополнительные операции (например, `get_min`)

Очередь на динамическом массиве

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> queue;

    queue.push(1);   // push
    int head = queue.front();  // head
    int tail = queue.back();   // tail
    std::cout << head << " " << tail << "\n";

    queue.pop();  // pop (удаляем с начала)
    std::cout << "size = " << queue.size() << "\n";
}
```

Циклическая очередь: идея

- Используется **массив фиксированной длины** n
- Два указателя:
  - `head` — начало очереди
  - `tail` — конец очереди
- При достижении конца массива индексы “зацикливаются”

Циклическая очередь: реализация

```
#include <iostream>
#include <vector>

const int n = 10;
std::vector<int> elements(n);
int head = 0;
int tail = 0;

bool empty() {
    return head == tail;
}

int size() {
    if (head > tail)
        return n - head + tail;
    else
        return tail - head;
}
```

Циклическая очередь: реализация

```
void push(int x) {
    if (size() != n - 1) {
        elements[tail] = x;
        tail = (tail + 1) % n;
    }
}

int pop() {
    if (empty())
        return -1;

    int x = elements[head];
    head = (head + 1) % n;
    return x;
}
```

Циклическая очередь: пример использования

```
int main() {
    std::vector<int> elems_to_add = {1, 4, 2, 3, 5, 2};
    for (int elem : elems_to_add) {
        push(elem);
    }

    std::cout << "size = " << size() << "\n\n";

    while (!empty()) {
        std::cout << pop() << " ";
    }
    std::cout << "\n";
}
```

- После добавления элементов — `size()` совпадает с их количеством
- При извлечении элементы выходят в порядке добавления
- Когда `head` достигает конца массива, он возвращается к началу

Очередь на двух стеках: идея

- `push` выполняется на первом стеке — $O(1)$
- `pop` — из второго стека, если он не пуст — $O(1)$
- Если второй стек пуст — перекладываем все элементы из первого — $O(n)$
- `size` поддерживаем отдельно
- Можно добавить `get_min` как `min(stack_1, stack_2)`

Очередь на двух стеках: реализация

```
#include <iostream>
#include <stack>
#include <algorithm>

std::stack<int> s1, s2;

void push(int x)
    s1.push(x);

int size()
    return s1.size() + s2.size();

int get_min() {
    if (s1.empty() && s2.empty()) return -1;
    int min1 = s1.empty() ? INT_MAX : s1.top();
    int min2 = s2.empty() ? INT_MAX : s2.top();
    return std::min(min1, min2);
}
```

Очередь на двух стеках: реализация

```
int pop() {
    if (s2.empty()) {
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
    }

    if (s2.empty()) return -1;

    int val = s2.top();
    s2.pop();
    return val;
}
```

Очередь на двух стеках: пример

```
int main() {
    push(5);
    push(2);
    push(8);
    std::cout << "size = " << size() << "\n";

    std::cout << pop() << "\n";  // 5
    std::cout << pop() << "\n";  // 2

    push(1);
    std::cout << pop() << "\n";  // 8
    std::cout << pop() << "\n";  // 1
}
```

- Среднее время работы pop — амортизированное $O(1)$
- Каждый элемент перекладывается **ровно один раз**

Очередь: итоги

- Очередь — структура FIFO: первый пришёл — первый ушёл
- Основные операции: `push`, `pop`, `head`, `tail`, `size`
- Основные реализации:
  - динамический массив (простая)
  - циклический буфер (эффективная)
  - два стека (функциональная и расширяемая)
- Выбор реализации зависит от задач и ограничений по памяти