Стек и память. Функции

Denis Bakin

Повторение — ссылки и указатели

- Что такое ссылка?

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти
  - Хранит адрес начала объекта

Повторение — ссылки и указатели

- Что такое ссылка?
- Ссылка — псевдоним существующей переменной
  - Не выделяет память
  - Используется как оригинал
  - `T&`: например, `int&`, `std::vector<int>&`
- Что такое указатель?
- Указатель — адрес объекта в памяти
  - Хранит адрес начала объекта
  - На 64-битных системах занимает **8 байт**

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 $\rightarrow$ 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

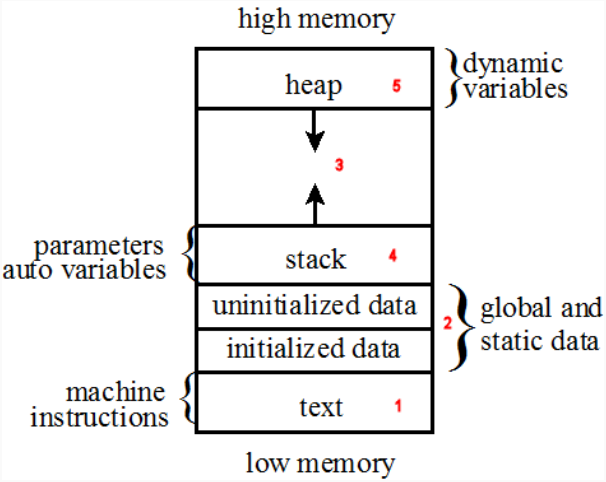
- Задача: поддерживать `push`, `pop`, `top` и `get_min()` за `O(1)`
- Идея: дополнительный стек с текущими минимумами
- при `push(x)` — если `x <= current_min` — `push` и в дополнительный стек
- при `pop()` — если удаляемый элемент == вершине стека минимумов — `pop` и там
- `get_min()` — вершина дополнительного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

Использование памяти

- На стеке: адрес возврата, сохранённые регистры, часть аргументов, локальные переменные
- В статической памяти: глобальные и статические переменные
- В динамической памяти (куче): объекты, выделенные с помощью `new`



Динамическая память (heap)

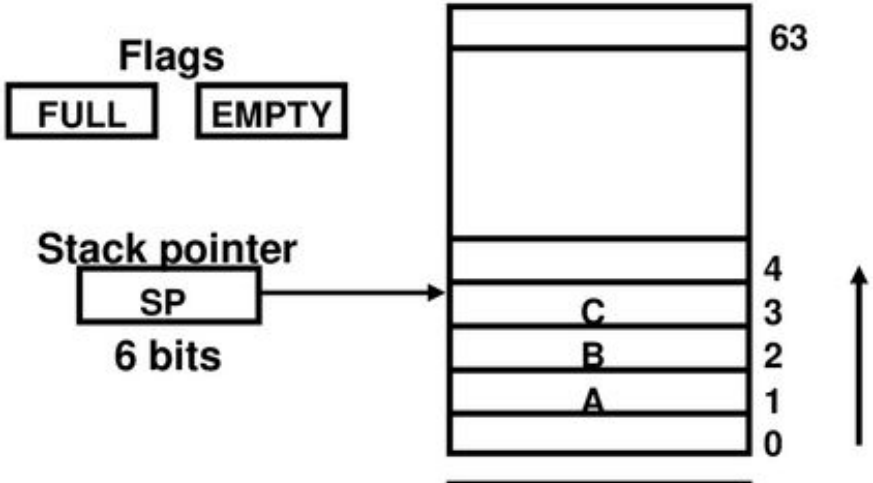
- Выделяется и освобождается во время выполнения
- В C++: `new` / `delete`, `new[]` / `delete[]`
- Когда использовать:
  - размер неизвестен на этапе компиляции
  - объект должен жить дольше области видимости
  - слишком большой для стека
- Ключевые правила:
  - каждому `new` — один `delete`
  - каждому `new[]` — один `delete[]`
  - не удалять дважды, не использовать после `delete`
  - утечки памяти — когда указатель потерян, а память не освобождена
  - для отладки — санитайзеры (`-fsanitize=address`)

Динамическая память (heap)

```
int main() {
    // создаем локальные переменные a и b на стеке, именно в таком порядке
    int a = 1;
    int b = 2;

    // указатель -- на стеке. int -- на куче
    int* p = new int(a);
    // использование *p

    std::cout << p << std::endl; // 0x7ffeedcba098
    std::cout << *p << std::endl; // 1

    delete p; // освобождение памяти
}
// здесь b и a будут автоматически деаллоцированы при выходе из main
// именно в таком порядке из-за LIFO
```

Реализация стека с помощью динамической памяти

- Подход: массив `T[]` в куче + указатель на вершину (`top_ptr`)



Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента
- `push(elem)` — сохранить по `stack_top`, инкрементировать `stack_top`

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента
- `push(elem)` — сохранить по `stack_top`, инкрементировать `stack_top`
- `pop()` — декрементировать `stack_top`

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента
- `push(elem)` — сохранить по `stack_top`, инкрементировать `stack_top`
- `pop()` — декрементировать `stack_top`
- `top()` — вернуть `*(stack_top - 1)`

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента
- `push(elem)` — сохранить по `stack_top`, инкрементировать `stack_top`
- `pop()` — декрементировать `stack_top`
- `top()` — вернуть `*(stack_top - 1)`
- `empty()` — `stack_top == stack_begin`

Реализация стека с помощью динамической памяти

Идея работы:

- выделяем `new T[capacity]`
- `stack_top` указывает на позицию следующего свободного элемента
- `push(elem)` — сохранить по `stack_top`, инкрементировать `stack_top`
- `pop()` — декрементировать `stack_top`
- `top()` — вернуть `*(stack_top - 1)`
- `empty()` — `stack_top == stack_begin`
- в конце — `delete[] stack_begin`

Что такое функция

- Функция — **именованный блок кода**, который можно вызвать
- Принимает аргументы
- Возвращает значение (через `return`)
- Повышает читаемость и повторное использование

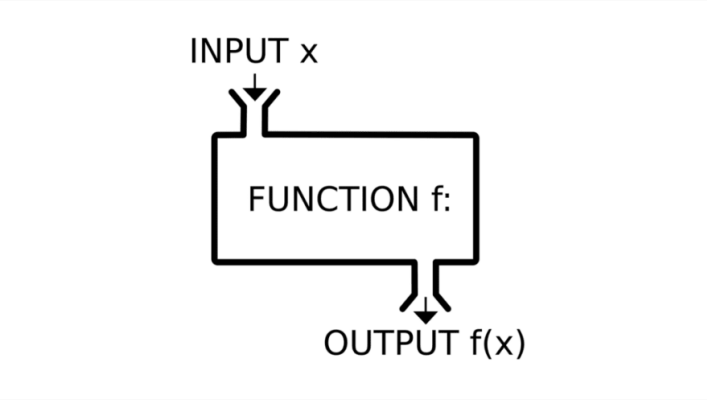


Figure 2: Иллюстрация работы функции

Что такое функция

- Функция — **именованный блок кода**, который можно вызвать
- Принимает аргументы
- Возвращает значение (через `return`)
- Повышает читаемость и повторное использование

```cpp
#include <iostream>

int getSum(int a, int b) {
    return a + b;
}

int main() {
    int first, second;
    std::cin >> first >> second;
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
}
```

Функции с условиями

- Вызов функции с несколькими аргументами разных типов
- `void` — тип функции, которая “ничего не возвращает”
- такие функции называются процедурами
- Внутри можно использовать `return` для выхода

Функции с условиями

Пусть есть некоторая задача с проверками:

- подсчитать количество чётных чисел
- проверить последние цифры суммы
- вывести вердикт по результатам проверок

```
// #include ...

void outputVerdict(size_t cnt, size_t t, uint64_t last_digits, uint64_t required_last_digits)
    if (last_digits != required_last_digits && cnt > t) {
        std::cout << "all checks failed!\n";
    } else if (last_digits != required_last_digits) {
        std::cout << "last digits do not match!\n";
    } else if (cnt > t) {
        std::cout << "too many even numbers!\n";
    } else {
        std::cout << "OK!\n";
    }
}
```

Упрощённый вариант с return

- Можно завершать выполнение функции сразу после вывода
- Избегаем каскада `if -- else if`

```
void outputVerdict(size_t cnt, size_t t, uint64_t last_digits, uint64_t required_last_digits)
    if (last_digits != required_last_digits && cnt > t) {
        std::cout << "all checks failed!\n";
        return;
    }
    if (last_digits != required_last_digits) {
        std::cout << "last digits do not match!\n";
        return;
    }
    if (cnt > t) {
        std::cout << "too many even numbers!\n";
        return;
    }
    std::cout << "OK!\n";
}
```

Зачем нужны функции?

Зачем нужны функции?

- Переиспользование кода

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)
- Разделение логики на блоки

Зачем нужны функции?

- Переиспользование кода
- Повышение читаемости. Например, названия отражают смысл (`isEven`, `getSum`)
- Разделение логики на блоки
- Упрощение отладки и тестирования

Аргументы функций

- Аргументы — данные, от которых зависит поведение функции
- Их можно передавать:
  - **по значению** – есть копирование, нельзя менять значение
  - **по ссылке** – нет копирования, можно менять значение

Передача по значению

- Значения копируются
- Изменения не влияют на внешние переменные
- Подходит для небольших типов (`int`, `char`, `bool`)

```cpp
#include <iostream>

int getSum(int a, int b) {
    a = 2;
    return a + b;
}

int main() {
    int first, second;
    std::cin >> first >> second;  // 6 7
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
    std::cout << first << ' ' << second << '\n';  // 6 7
}
```

Передача по ссылке

- Копирования **нет**
- Изменение аргумента внутри функции изменяет оригинал
- Используется для больших структур и при необходимости изменять внешние переменные

```
#include <iostream>

int getSum(int& a, int& b) {
    a = 2;
    return a + b;
}

int main() {
    int first, second;
    std::cin >> first >> second;  // 5 1
    int result = getSum(first, second);
    std::cout << "Sum is " << result << '\n';
    std::cout << first << ' ' << second << '\n';  // 2 1
}
```

Аргумент по ссылке — без копирования

```cpp
#include <iostream>
#include <string>

size_t countChar(std::string& line, char chr_to_count) {
    size_t cnt = 0;
    for (const char& chr : line) {
        if (chr == chr_to_count) {
            ++cnt;
        }
    }
    return cnt;
}

int main() {
    std::string input_line;
    std::getline(std::cin, input_line);
    std::cout << countChar(input_line, 'a') << '\n';
}
```

Аргумент по ссылке — изменение внешней переменной

```cpp
#include <iostream>
#include <string>

void addToString(std::string& line, char chr_to_fill, size_t num) {
    for (size_t i = 0; i < num; ++i) {
        line += chr_to_fill;
    }
}
```

- Функция изменяет строку из вызывающего кода
- Передача по ссылке обязательна, иначе изменения не сохранятся

Необязательные аргументы

- Можно задавать **значения по умолчанию**
- Аргументы передаются **позиционно**
- Нельзя пропускать средние аргументы

```
#include <iostream>
#include <string>

void printMessage(const std::string& message, char border_char = '*', int repeat_count = 3) {
    for (int i = 0; i < repeat_count; ++i)
        std::cout << border_char;
    std::cout << " " << message << " ";
    for (int i = 0; i < repeat_count; ++i)
        std::cout << border_char;
    std::cout << '\n';
}
```

Пример вызовов с необязательными аргументами

```cpp
#include <iostream>
#include <string>

void printMessage(const std::string& message, char border_char = '*', int repeat_count = 3);

...

int main() {
    printMessage("Some line of text");
    printMessage("Some line of text", '#');
    printMessage("Some line of text", '=', 5);
    printMessage("Some line of text", 'a', 0);
}
```

- Значения по умолчанию — удобный способ сократить вызовы
- Используются для часто повторяющихся настроек
- Нет именованных аргументов, как в Python

Возвращаемое значение

- При `return` **копирования не происходит** (copy elision)
- Компилятор оптимизирует размещение результата

```cpp
#include <iostream>
#include <string>

uint64_t calcSum(uint64_t a, uint64_t b) {
    uint64_t ret_value = a + b; // с copy elision &result == &ret_value
    ++ret_value;
    return ret_value; // нет копирования, только выход из функции
}

int main() {
    uint64_t first, second;
    std::cin >> first >> second;
    uint64_t result = calcSum(first, second); // с copy elision &result == &ret_value
    std::cout << "Result is " << result << '\n';
}
```

Перегрузка функций

- Одинаковое имя
- Разные аргументы (тип, количество, порядок)
- Повышает читаемость и гибкость

```cpp
#include <iostream>
#include <string>

void logger(const std::string& message) {
    std::cout << "[INFO]: " << message << std::endl;
}

void logger(int errorCode, const std::string& message) {
    std::cout << "[ERROR " << errorCode << "]: " << message << std::endl;
}

void logger(const std::string& message, const std::string& severity) {
    std::cout << "[" << severity << "]: " << message << std::endl;
}
```

Пример перегрузок в действии

```
void logger(const std::string& message);
void logger(int errorCode, const std::string& message);
void logger(const std::string& message, const std::string& severity);

int main() {
    logger("System started successfully.");
    logger(404, "Resource not found.");
    logger("Disk space running low", "WARNING");
}
```

Вывод:

```
[INFO]: System started successfully.
[ERROR 404]: Resource not found.
[WARNING]: Disk space running low
```

Лямбда функции — основа

- **Анонимные функции** без имени
- Можно хранить в переменной
- Синтаксис:

```
[= or &](args) { ... };
```

```
auto adder = [](int a, int b){
    return a + b;
};
```

Пример лямбда-функции

```cpp
#include <iostream>

int main() {
    int first, second;
    std::cin >> first >> second;
    auto adder = [](int a, int b){
        return a + b;
    };
    int result = adder(first, second);
    std::cout << "Sum is " << result << '\n';
}
```

Захват контекста (capture)

- [] — ничего не видно

Захват контекста (capture)

- [] — ничего не видно
- [=] — по **значению**, контекст копируется

Захват контекста (capture)

- `[]` — ничего не видно
- `[=]` — по **значению**, контекст копируется
- `[&]` — по **ссылке**, контекст доступен для изменения и без копирования

Захват по значению и по ссылке

Захват по ссылке

```
int increment_by = 15;
auto ref_incrementer = [&](int num){
    return num + increment_by;
};
cout << ref_incrementer(15) << '\n'; // 30
increment_by = 10;
cout << ref_incrementer(15) << '\n'; // 25
```

Захват по значению

```
int increment_by = 15;
auto val_incrementer = [=](int num){
    return num + increment_by;
};
cout << val_incrementer(15) << '\n'; // 30
increment_by = 10;
cout << val_incrementer(15) << '\n'; // 30
```

Функции высшего порядка

- Функция принимает или возвращает **другую функцию**
- Используется `std::function<T(Args ...)>` для типов функций

```
#include <iostream>
#include <functional>

void outputVerdict(int number, std::function<bool(int)> checker) {
    if (checker(number)) {
        std::cout  << "Checker returned true for this value!" << std::endl;
    } else {
        std::cout  << "Checker returned false for this value!" << std::endl;
    }
}
```

Пример функции высшего порядка

```
int main() {
    int a = 1;
    int b = 2;
    std::function<bool(int)> is_even = [](int number) {
        return number % 2 == 0;
    };

    outputVerdict(a, is_even);
    outputVerdict(b, is_even);
}
```

Итоги

- Функции делают код понятным, модульным и масшабируемым
- Передача по ссылке экономит память
- Необязательные аргументы делают интерфейс гибким
- Перегрузка упрощает использование одной логики для разных наборов аргументов