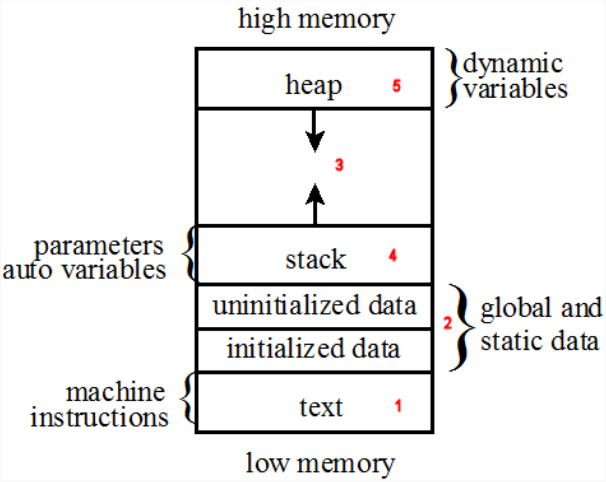
Динамическая память. Основы ООП. Связные списки

Denis Bakin

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
```

Структуры

- Позволяют **объединять логически связанные данные** под одним именем
- Можно создавать **новые типы данных**
- Каждый объект структуры содержит **поля** (переменные)

Пример структуры Person

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

struct Person {
    std::string name;
    int height;
    int age;
    bool expelled;
};
```

- Каждое поле имеет собственный тип
- Можно инициализировать по-разному

Использование структуры Person

```cpp
int main() {
    Person person1{"Алексей", 180, 22, false};
    Person person2{"Мария", 165, 20, true};
    Person person3{"Иван", 175, 23, false};
    Person person4{"Екатерина", 170, 21, false};
    Person person5{"Петр", 185, 24, true};

    std::vector<Person> people = {person1, person2, person3, person4, person5};

    for (const Person& person : people) {
        std::cout << "Имя: " << person.name << "\n"
                  << "Рост: " << person.height << "\n"
                  << "Возраст: " << person.age << "\n"
                  << "Отчислен: " << (person.expelled ? "Да" : "Нет") << "\n"
                  << "-----------------\n";
    }
}
```

Структура Point

- Частный случай — хранение координат точки в 3D
- Структура упрощает передачу данных в функции

```
struct Point {
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;
};
```

Работа с точками: расстояние между двумя точками

- как найти расстояние между двумя точками в пространстве?

Работа с точками: расстояние между двумя точками

- как найти расстояние между двумя точками в пространстве?

```cpp
#include <cmath>

double getDistance(const Point& first, const Point& second) {
    double sq = std::pow(first.x - second.x, 2)
              + std::pow(first.y - second.y, 2)
              + std::pow(first.z - second.z, 2);
    return std::sqrt(sq);
}
```

Работа с точками: найти ближайшую точку

- как найти ближайшую точку из набора к заданной?

Работа с точками: найти ближайшую точку

- как найти ближайшую точку из набора к заданной?

```cpp
#include <vector>

Point findClosestPoint(const Point& target, const std::vector<Point>& points) {
    Point closest = points[0];
    double minDist = getDistance(closest, target);
    for (const Point& p : points) {
        double d = getDistance(p, target);
        if (d < minDist) {
            minDist = d;
            closest = p;
        }
    }
    return closest;
}
```

Пример использования Point

```
int main() {
    Point p1(1.0, 2.0, 3.0);
    Point p2(4.0, 5.0, 6.0);
    Point p3(7.0, 8.0, 9.0);
    Point p4(2.0, 2.0, 2.0);
    std::vector<Point> points = {p1, p2, p3};

    std::cout << "Расстояние: " << getDistance(p1, p2) << '\n';
    Point closest = findClosestPoint(p4, points);
    std::cout << "Ближайшая: (" << closest.x << ", "
              << closest.y << ", " << closest.z << ")\n";
}
```

Методы структур

- Что такое метод структуры (member function):
  - функция, объявленная внутри `struct`/`class` и имеющая доступ к полям `this`
  - удобный способ связать поведение с данными
- Альтернатива — свободные функции, принимающие объект в аргументах

Методы структур

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

struct Person {
    std::string name;
    int age;
};

// свободная функция, принимает объект
void printPerson(const Person& p) {
    std::cout << p.name << " (" << p.age << ")\n";
}

int main() {
    Person a{"Алексей", 22};
    printPerson(a);
}
```

Методы структур

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

struct Person {
    std::string name;
    int age;

    // метод структуры
    void print() const {
        std::cout << name << " (" << age << ")\n";
    }
};

int main() {
    Person a{"Алексей", 22};
    a.print();
}
```

Связные списки

- Связный список — набор узлов, каждый содержит данные и ссылки на соседей
- С помощью списков можно реализовать стек и очередь
- Виды:
  - односвязный (next)
  - двусвязный (next + prev)
  - циклический (последний → первый)

Figure 2: Односвязный список

- какую информацию должен хранить каждый узел односвязного списка?

Связные списки

Односвязный список — базовый узел

```
struct Node {
    int value = 0;
    Node* next = nullptr;
    Node(int v) : value(v) {}
};
```

- `next` — ссылка на следующий узел (или `nullptr`)
- какую информацию должен хранить каждый узел двусвязного списка?

Связные списки

Двусвязный список — базовый узел

```
struct Node {
    int value = 0;
    Node* next = nullptr;
    Node* prev = nullptr;
    Node(int v) : value(v) {}
};
```

Figure 3: Двусвязный список

Связные списки

Вставка (insertAfter)

- Добавление элемента `thatElement` после `thisElement`

Вставка (insertAfter)

- Добавление элемента thatElement после thisElement

```
void insertAfter(Node* thisElement, Node* thatElement) {
    if (!thisElement) return;
    thatElement->next = thisElement->next;
    thisElement->next = thatElement;
}
```

- Для вставки в голову: менять head (см. removeHead/insertHead)

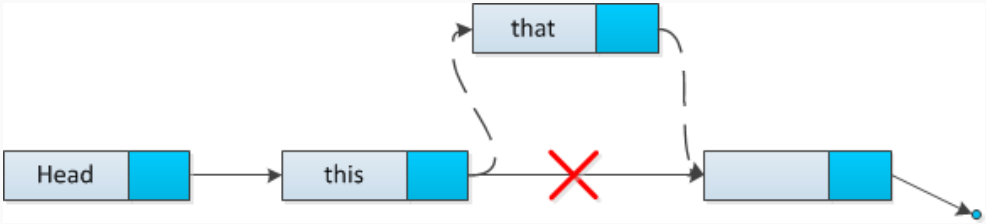


Figure 4: Вставка после

Связные списки

Поиск (search)

- Идём от головы, сравниваем значения:

```
Node* search(Node* head, int value) {
    Node* node = head;
    while (node != nullptr && node->value != value) {
        node = node->next;
    }
    return node; // nullptr если не найден
}
```

- Сложность: $O(n)$

Связные списки

Удаление головы (removeHead)

```
void removeHead(Node*& head) {
    if (head != nullptr) {
        Node* tmp = head;
        head = head->next;
        delete tmp;
    }
}
```

- При удалении обязательно delete — иначе утечка памяти

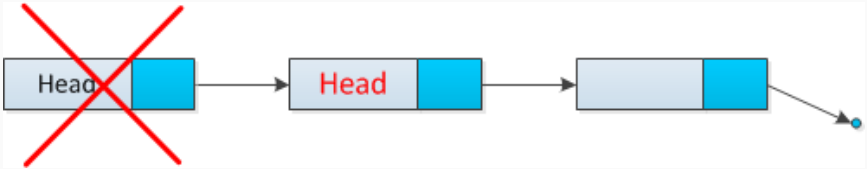


Figure 5: Удаление головы

Связные списки

Удаление после элемента (removeAfter)

```
void removeAfter(Node* thisElement) {
    if (thisElement == nullptr || thisElement->next == nullptr) return;
    Node* tmp = thisElement->next;
    thisElement->next = tmp->next;
    delete tmp;
}
```

- Не забывать про `nullptr`-проверки

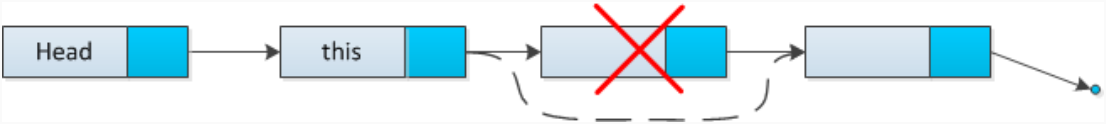


Figure 6: Удаление после

Связные списки

Двусвязный список — узел

```
struct DNode {
    int value;
    DNode* prev;
    DNode* next;
    DNode(int v) : value(v), prev(nullptr), next(nullptr) {}
};
```

- Удобнее удалять и вставлять произвольно (есть `prev`)
- Нужна аккуратная работа с краями списка

Связные списки

Циклический список

- В циклическом списке `last->next == head`
- Полезно для круговых буферов и игр (круговой проход)
- Нужно внимательно обрабатывать условия остановки при обходе

Figure 7: Циклический список

Поиск цикла: алгоритм Флойда (tortoise & hare)

Обнаружение цикла

- Если встретились — есть цикл; если `hare` дошёл до конца — цикла нет
- Сложность: $O(n)$, память: $O(1)$

Поиск начала цикла: алгоритм Флойда (tortoise & hare)

Поиск начала цикла

- работает за $O(n)$, как только мы разметили точку на цикле
- точка встречи `pointMeeting` найдена алгоритмом Флойда
- запустить один указатель из `head`, другой — из `pointMeeting`, двигать оба на 1 — точка их встречи = начало цикла

Связные списки

Разворот списка — итеративный метод

```
Node* reverseIterative(Node* head) {
    Node* prev = nullptr;
    Node* curr = head;
    while (curr != nullptr) {
        Node* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev; // новая голова
}
```

- Без рекурсии, память $O(1)$
- Сложность: $O(n)$

Связные списки

Практические советы и ошибки

- Всегда проверять `nullptr` перед доступом к `->next`
- Не забывать вызывать `delete` для удалённых узлов — чтобы избежать утечек
- Быть осторожным с двойным удалением (double free)

Связные списки

- Связные списки — гибкая структура для динамического хранения и манипуляций
- Односвязный проще, двусвязный удобнее для операций удаления/вставки в середине
- Алгоритмы: вставка/удаление/поиск — $O(n)$; разворот — $O(n)$; поиск цикла — $O(n)$