



Лекция 4. Ручное управление памятью

Илья Макаров

CS центр

28 сентября 2021

Санкт-Петербург

Еще раз про указатели

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

```
int i = 3; // переменная типа int  
int * p = 0; // указатель на переменную типа int
```

- Нулевому указателю (`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
```

Указатели и const

- Константный указатель – это переменная, хранящая адрес некоторой ячейки памяти, который нельзя изменить.
- Указатель на константу – это переменная, хранящая адрес некоторой ячейки памяти. Данные в этой ячейки нельзя изменить.
- Модификатор – часть типа.

```
int i = 3;  
int * const p1 = &i; // константный указатель  
int const * p2 = &i; // указатель на константу  
const int * p3 = &i; // указатель на константу
```

```
p1 = nullptr; // error  
*p1 = 10; // ok  
p2 = nullptr; // ok  
*p2 = 10; // error
```

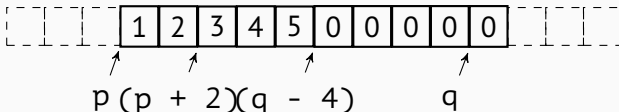


Связь массивов и указателей

- Указатели позволяют передвигаться по массивам.
- Для этого используется арифметика указателей:

```
int m[10] = {1, 2, 3, 4, 5};  
int * p = &m[0]; // адрес начала массива  
int * q = &m[9]; // адрес последнего элемента
```

- $(p + k)$ – сдвиг на k ячеек типа `int` вправо.
- $(p - k)$ – сдвиг на k ячеек типа `int` влево.
- $(q - p)$ – количество ячеек между указателями.
- $p[k]$ эквивалентно $*(p + k)$.



Примеры

Заполнение массива:

```
int m[10] = {}; // изначально заполнен нулями

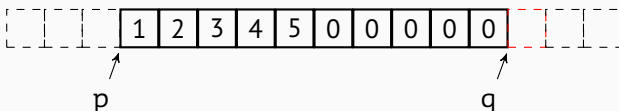
for (int * p = m ; p <= m + 9; ++p) {
    *p = (p - m) + 1;
} // Массив заполнен числами от 1 до 10
```

Передача массива в функцию:

```
int max_element(int * m, int size) {
    int max = *m;
    for (int i = 1; i < size; ++i) {
        if (m[i] > max) { max = m[i]; }
    }
    return max;
}
```

Два способа передачи массива

```
bool contains(int * m, int size, int value) {  
    for (int i = 0; i != size; ++i) {  
        if (m[i] == value) { return true; }  
    }  
    return false;  
}  
  
bool contains(int * p, int * q, int value) {  
    for (; p != q; ++p) {  
        if (*p == value) { return true; }  
    }  
    return false;  
}
```



Возрат указателя из функции

Функция для поиска максимума в массиве:

```
int * max_element(int * p, int * q) {  
    int * pmax = p;  
    for (; p != q; ++p) {  
        if (*p > *pmax) { pmax = p; }  
    }  
    return pmax;  
}
```

```
int m[10] = {...};  
int * pmax = max_element(m, m + 10);  
cout << "Maximum = " << *pmax << endl;
```

Возрат значения через указатель

Функция для поиска максимума в массиве:

```
bool max_element(int * p, int * q, int * res) {  
    if (p == q) { return false; }  
    *res = *p;  
    for (; p != q; ++p) {  
        if (*p > *res) { *res = *p; }  
    }  
    return true;  
}
```

```
int m[10] = {...};  
int max = 0;  
if (max_element(m, m + 10, &max)) {  
    cout << "Maximum = " << max << endl;  
}
```


Возрат значения через указатель на указатель

Функция для поиска максимума в массиве:

```
bool max_element(int * p, int * q, int ** res) {  
    if (p == q)  
        return false;  
    *res = p;  
    for (; p != q; ++p)  
        if (*p > **res)  
            *res = p;  
    return true;  
}
```

```
int m[10] = {...};  
int * pmax = 0;  
if (max_element(m, m + 10, &pmax))  
    cout << "Maximum = " << *pmax << endl;
```

Недостатки указателей

- Использование указателей синтаксически загрязняет код и усложняет его понимание. (Приходится использовать операторы * и &.)
- Указатели могут быть неинициализированными (некорректный код).
- Указатель может быть нулевым (корректный код), а значит указатель нужно проверять на равенство нулю.
- Арифметика указателей может сделать из корректного указателя некорректный (легко промахнуться).
- Тяжелее отследить время жизни указателей.



Еще раз про различия ссылок и указателей

- Ссылка не может быть неинициализированной.

```
int * p; // OK  
int & l; // ошибка
```

- У ссылки нет нулевого значения.

```
int * p = 0; // OK  
int & l = 0; // ошибка
```

- Ссылку нельзя переписать:

```
int a = 10, b = 20;  
int * p = &a; // p указывает на a  
p = &b;       // p указывает на b  
int & l = a;  // l ссылается на a  
l = b;       // a присваивается значение b
```

Еще раз про различия ссылок и указателей

- Нельзя получить адрес ссылки или ссылку на ссылку.

```
int a = 10;  
int * p = &a; // p указывает на a  
int ** pp = &p; // pp указывает на переменную p  
int & l = a; // l ссылается на a  
int * pl = &l; // pl указывает на переменную a  
int && ll = l; // ошибка
```

- Нельзя создавать массивы ссылок.

```
int * mp[10] = {}; // массив указателей на int  
int & ml[10] = {}; // ошибка
```

- Для ссылок нет арифметики.

Ссылки и rvalue

Ссылки могут указывать только на lvalue.

```
int a = 10, b = 20;  
int m[10] = {1,2,3,4,5,5,4,3,2,1};  
int & l1 = a;           // OK  
int & l2 = a + b;        // ошибка  
int & l3 = *(m + a / 2); // OK  
int & l4 = *(m + a / 2) + 1; // ошибка  
int & l5 = (a + b > 10) ? a : b; // OK
```

Зачем нужна динамическая память?

- Стек программы ограничен. Он не предназначен для хранения больших объемов данных.

```
// Не умещается на стек  
double m[10000000] = {}; // 80 Мб
```

- Время жизни локальных переменных ограничено временем работы функции.
- Динамическая память выделяется в сегменте данных.
- Структура, отвечающая за выделение дополнительной памяти, называется **кучей** (не нужно путать с одноимённой структурой данных).
- Выделение и освобождение памяти *управляется вручную*.

Выделение памяти в стиле C

- Стандартная библиотека `cstdlib` предоставляет четыре функции для управления памятью:

```
void * malloc(size_t size);  
void * calloc(size_t nmemb, size_t size);  
void * realloc(void * ptr, size_t size);  
void free(void * ptr);
```

- `size_t` — специальный целочисленный беззнаковый тип, может вместить в себя размер любого типа в байтах.
- Тип `size_t` используется для указания размеров типов данных, для индексации массивов и пр.
- `void *` — это указатель на нетипизированную память (раньше для этого использовалось `char *`).

Выделение памяти в стиле C

- Функции для управления памятью в стиле C:

```
void * malloc(size_t size);  
void * calloc(size_t nmemb, size_t size);  
void * realloc(void * ptr, size_t size);  
void free(void * ptr);
```

- **malloc** – выделяет область памяти размера \geq **size**. Данные не инициализируются.
- **calloc** – выделяет массив из **nmemb** элементов размера **size**. Данные инициализируются нулём.
- **realloc** – изменяет размер области памяти по указателю **ptr** на **size** (если возможно, то это делается на месте).
- **free** – освобождает область памяти, ранее выделенную одной из функций **malloc/calloc/realloc**.



Выделение памяти в стиле C

- Для указания размера типа используется оператор `sizeof`.

```
// создание массива из 1000 int
int * m = (int *)malloc(1000 * sizeof(int));
m[10] = 10;
```

```
// изменение размера массива до 2000
m = (int *)realloc(m, 2000 * sizeof(int));
```

```
// освобождение массива
free(m);
```

```
// создание массива нулей
m = (int *)calloc(3000, sizeof(int));
free(m);
m = 0;
```

Выделение памяти в стиле C++

- Язык C++ предоставляет два набора операторов для выделения памяти:
 1. `new` и `delete` — для одиночных значений,
 2. `new []` и `delete []` — для массивов.
- Версия оператора `delete` должна соответствовать версии оператора `new`.

```
// выделение памяти под один int со значением 5  
int * m = new int(5);  
delete m; // освобождение памяти
```

```
// создание массива нулей  
m = new int[1000](); // () означает обнуление  
delete [] m; // освобождение памяти
```



Типичные проблемы при работе с памятью

- Проблемы производительности: создание переменной на стеке намного “дешевле” выделения для неё динамической памяти.
- Проблема фрагментации: выделение большого количества небольших сегментов способствует фрагментации памяти.
- Утечки памяти:

```
// создание массива из 1000 int
int * m = new int[1000];

// создание массива из 2000 int
m = new int[2000]; // утечка памяти

// Не вызван delete [] m, утечка памяти
```

Типичные проблемы при работе с памятью

- Неправильное освобождение памяти.

```
int * m1 = new int[1000];  
delete m1; // должно быть delete [] m1  
  
int * p = new int(0);  
free(p); // совмещение функций C++ и C  
  
int * q1 = (int *)malloc(sizeof(int));  
free(q1);  
free(q1); // двойное удаление  
  
int * q2 = (int *)malloc(sizeof(int));  
free(q2);  
q2 = 0; // обнуляем указатель  
free(q2); // правильно работает для q2 = 0
```



Многомерные встроенные массивы

- C++ позволяет определять многомерные массивы:

```
int m2d[2][3] = { {1, 2, 3}, {4, 5, 6} };  
for( size_t i = 0; i != 2; ++i ) {  
    for( size_t j = 0; j != 3; ++j ) {  
        cout << m2d[i][j] << ' ';  
    }  
    cout << endl;  
}
```

- Элементы m2d располагаются в памяти “по строчкам”.
- Размерность массивов может быть любой, но на практике редко используют массивы размерности > 4 .

```
int m4d[2][3][4][5] = {};
```

Динамические массивы

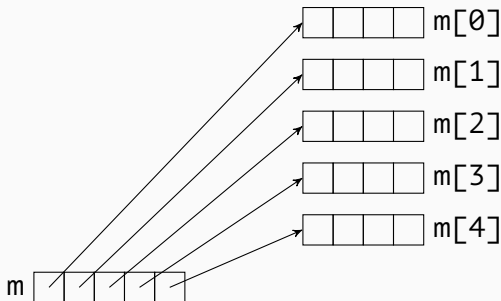
- Для выделения одномерных динамических массивов обычно используется оператор `new []`.

```
int * m1d = new int[100];
```

- Какой тип должен быть у указателя на двумерный динамический массив?
 - Пусть `m` – указатель на двумерный массив типа `int`.
 - Значит `m[i][j]` имеет тип `int` (точнее `int &`).
 - $m[i][j] \Leftrightarrow *(m[i] + j)$, т.е. тип `m[i]` – `int *`.
 - аналогично, $m[i] \Leftrightarrow *(m + i)$, т.е. тип `m` – `int **`.
- Чему соответствует значение `m[i]`?
Это адрес строки с номером `i`.
- Чему соответствует значение `m`?
Это адрес массива с указателями на строки.

Двумерные массивы

Давайте рассмотрим создание массива 5×4 .



```
int ** m = new int * [5];  
for (size_t i = 0; i != 5; ++i)  
    m[i] = new int[4];
```

Двумерные массивы

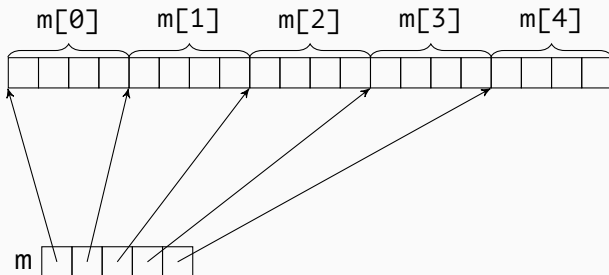
Выделение и освобождение двумерного массива размера $a \times b$.

```
int ** create_array2d(size_t a, size_t b) {  
    int ** m = new int *[a];  
    for (size_t i = 0; i != a; ++i)  
        m[i] = new int[b];  
    return m;  
}  
  
void free_array2d(int ** m, size_t a, size_t b) {  
    for (size_t i = 0; i != a; ++i)  
        delete [] m[i];  
    delete [] m;  
}
```

При создании массива оператор `new` вызывается $(a + 1)$ раз.

Двумерные массивы: эффективная схема

Рассмотрим эффективное создание массива 5×4 .



```
int ** m = new int * [5];  
m[0] = new int[5 * 4];  
for (size_t i = 1; i != 5; ++i)  
    m[i] = m[i - 1] + 4;
```

Двумерные массивы: эффективная схема

Эффективное выделение и освобождение двумерного массива размера $a \times b$.

```
int ** create_array2d(size_t a, size_t b) {  
    int ** m = new int *[a];  
    m[0] = new int[a * b];  
    for (size_t i = 1; i != a; ++i)  
        m[i] = m[i - 1] + b;  
    return m;  
}  
  
void free_array2d(int ** m, size_t a, size_t b) {  
    delete [] m[0];  
    delete [] m;  
}
```

При создании массива оператор `new` вызывается 2 раза.

Умные указатели

1. Идиомы RAII (Resource Acquisition Is Initialization): время жизни ресурса связано с временем жизни объекта.
 - Получение ресурса в конструкторе.
 - Освобождение ресурса в деструкторе.
2. Основные области использования RAII:
 - для управления памятью,
 - для открытия файлов или устройств,
 - для критических секций при параллельном исполнении кода.
3. Умные указатели — объекты, инкапсулирующие владение памятью. Синтаксически ведут себя так же, как и обычные указатели.

Основные стратегии

1. `scoped_ptr` — время жизни объекта ограничено временем жизни умного указателя.
2. `shared_ptr` — разделяемый объект, реализация с подсчётом ссылок.
3. `weak_ptr` — разделяемый объект, реализация с подсчётом ссылок, слабая ссылка (используется вместе с `shared_ptr`).
4. `unique_ptr` — эксклюзивное владение объектом с передачей владения при перемещении.
5. `intrusive_ptr` — разделяемый объект, реализация самим внутри объекта.
6. `linked_ptr` — разделяемый объект, реализация списком указателей.

unique_ptr

- Определен в заголовочном файле `<memory>`.
- Для передачи и возврата указателей из функции.
- Владение эксклюзивно и передаётся при только при перемещении.

```
void foo(const std::unique_ptr<int> & ptr) {  
    std::cout << *ptr;  
}  
void bar(std::unique_ptr<int> && ptr) {  
    std::cout << *ptr;  
}  
void baz(std::unique_ptr<int> ptr) {  
    std::cout << *ptr;  
}
```

unique_ptr

```
int main() {  
    std::unique_ptr<int> ptr = std::make_unique<int>(10);  
  
    auto & ptr_ref = ptr; // ok  
    auto ptr_copy = ptr; // error  
  
    foo(ptr); // ok  
    bar(ptr); // error  
    bar(std::move(ptr)); // ok  
    baz(ptr); // error  
    baz(std::move(ptr)); // ok  
}
```

shared_ptr

- Для разделяемых объектов.
- Ведётся подсчёт ссылок.
- Нельзя вернуть владение объектом.

```
void foo(const std::shared_ptr<int> & ptr) {  
    std::cout << *ptr;  
}  
void bar(std::shared_ptr<int> && ptr) {  
    std::cout << *ptr;  
}  
void baz(std::shared_ptr<int> ptr) {  
    std::cout << *ptr;  
}
```

shared_ptr

```
int main() {  
    std::shared_ptr<int> ptr = std::make_shared<int>(10);  
  
    auto & ptr_ref = ptr; // no increment  
    auto ptr_copy = ptr; // increment  
  
    foo(ptr); // no increment  
    bar(ptr); // error  
    bar(std::move(ptr)); // no increment  
    baz(ptr) // increment  
    baz(std::move(ptr)); // no increment  
}
```


shared_ptr

```
void foo() {  
    // counter == 1  
    std::shared_ptr<int> ptr = std::make_shared<int>(10);  
  
    {  
        // counter == 2  
        auto ptr_copy = ptr;  
    } // counter == 1  
  
    {  
        // counter == 1  
        auto & ptr_ref = ptr;  
    } // counter == 1  
} // counter == 0
```

shared_ptr

```
void foo() {  
    // counter == 1  
    std::shared_ptr<int> ptr = std::make_shared<int>(10);  
  
    {  
        // counter == 1  
        auto ptr_other = std::move(ptr);  
    } // counter == 0  
}
```

weak_ptr

- Для использования вместе с shared_ptr.
- Слабая ссылка для исключения циклических зависимостей.
- Не владеет объектом.

```
void foo(std::weak_ptr & ptr_weak) {  
    if (auto ptr_locked = ptr_weak.lock()) {  
        // shared here  
    }  
}  
  
int main() {  
    std::shared_ptr<int> ptr = std::make_shared<int>(10);  
    foo(ptr); // implicit cast here  
}
```



Умные указатели и const

```
// константный указатель
const auto p1 = std::make_shared<int>(10);
// указатель на константу
auto p2 = std::make_shared<const int>(10);

p1 = nullptr; // error
*p1 = 10; // ok
p2 = nullptr; // ok
*p2 = 10; // error
```

Заключение

- Умные указатели намного удобнее ручного управления памятью.
- Для локальных объектов – `scoped_ptr` (подробнее на семинаре).
- Для разделяемых объектов – `shared_ptr`.
- В сильносвязанных системах рассмотрите возможность использовать `weak_ptr`.
- Используйте `intrusive_ptr` для тех объектов, которые сами управляют своим временем жизни.
- Прочитайте документацию по `shared_ptr` и `unique_ptr`.