Лекция 9. Динамическая память. Динамические массивы и матрицы

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

Содержание лекции

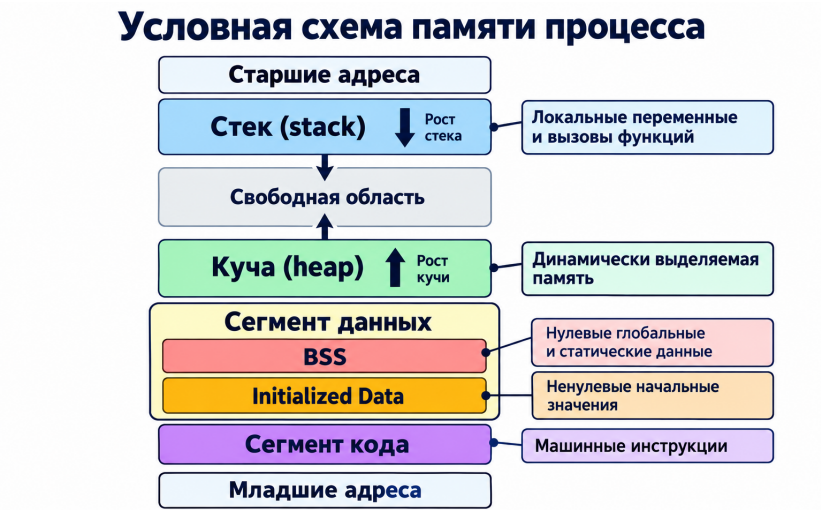
1. Области памяти: статическая, стек, куча.
2. Операторы `new` и `delete`.
3. Ошибки при работе с динамической памятью.
4. Динамические одномерные массивы.
5. Динамические двумерные массивы (матрицы).
6. Ступенчатые массивы.
7. Введение в умные указатели (концепция).
8. Итоги.

Области памяти в C++

Область	Назначение	Примеры
Стек (Stack)	Локальные переменные, параметры функций, адреса возврата. Растёт вниз.	`int x = 42;` (локальная)
Куча (Heap)	Динамическая память — выделяется через `new`, освобождается через `delete`. Растёт вверх.	`int* p = new int(5);`
— Статическая память (Сегмент данных, Data Segment) —		
BSS	Неинициализированные глобальные и статические переменные. Обнуляются при запуске.	`int g;` (глобальная), `static int s;`
Initialized Data	Инициализированные глобальные и статические переменные.	`int g = 5;`, `static int s = 10;`
Сегмент кода (Text Segment)	Машинный код программы. Обычно только для чтения.	Инструкции функций.

Стек и куча растут навстречу друг другу; между ними находится свободная область, которая может использоваться обеими динамическими структурами.

Схема памяти в C++



Стек vs Куча

Характеристика	Стек	Куча
Управление	Автоматическое	Ручное (`new`/`delete`)
Скорость	Быстрое выделение	Медленнее
Размер	Ограничен (обычно 1–8 МБ)	Ограничен адресным пространством и ресурсами системы
Время жизни	До конца блока	До явного `delete`
Ошибки	Переполнение стека	Утечки, висячие указатели

- Для больших данных или данных неизвестного размера используйте кучу.
- Для маленьких и кратковременных данных — стек.

Зачем нужна динамическая память?

- Размер данных неизвестен на этапе компиляции (зависит от ввода пользователя, файла и т.д.).
- Нужно создать структуру данных, размер которой меняется во время выполнения.
- Данные должны жить дольше, чем функция, в которой они созданы.
- Большие объёмы данных, которые не помещаются в стек.

Пример

Массив, размер которого вводит пользователь.

```
int n;
cin >> n;
int* arr = new int[n]; // размер становится известен только во время выполнения
```

Операторы new и delete (скалярные типы)

Выделение памяти

```
int* p = new int;       // выделение памяти для одного int
*p = 42;
cout << *p;             // 42
```

Освобождение памяти

```
delete p;               // освобождение памяти
p = nullptr;            // хороший тон - обнулить указатель
```

- new выделяет память в куче и возвращает указатель. Если память выделить не удалось, new бросает исключение std::bad_alloc.
- delete освобождает память, на которую указывает указатель.
- После delete указатель становится висячим — обнуляйте его!

Классификация ошибок при работе с динамической памятью

1. **Утечка памяти** — память выделена, но не освобождена.
2. **Двойное удаление** — `delete` вызван дважды для одного указателя.
3. **Разыменование висячего указателя** — доступ к памяти после `delete`.
4. **Потеря указателя** — указатель перезаписан, а старая память не освобождена.
5. **Несоответствие `new` и `delete`** — `delete` вместо `delete[]` и наоборот.
6. **Выход за границы** — доступ к элементам за пределами выделенного блока.

Утечка памяти — потеря указателя

```
int* p = new int(5); // выделили память
p = new int(10);     // перезаписали указатель, не удалив старый объект
// память с числом 5 потеряна - утечка!
delete p;            // удаляем только число 10
```

- Указатель p теперь указывает на новую память.
- Старая память (с числом 5) осталась недоступной.
- Утечка может накапливаться и приводить к исчерпанию памяти.

Утечка памяти — потеря указателя в цикле

```
for (int i = 0; i < 100; ++i) {
    int* p = new int(i);
    // забыли delete p
}
```

- Каждая итерация выделяет память, но не освобождает.
- После 100 итераций — 100 утечек.
- В больших программах это может привести к аварийному завершению.

Утечка памяти — потеря указателя в цикле

```
for (int i = 0; i < 100; ++i) {
    int* p = new int(i);
    // забыли delete p
}
```

- Каждая итерация выделяет память, но не освобождает.
- После 100 итераций — 100 утечек.
- В больших программах это может привести к аварийному завершению.

Решение: всегда освобождать память, когда она больше не нужна.

Утечка памяти — возврат указателя из функции

```
int* createArray(int size) {
    int* arr = new int[size];
    return arr; // память выделена, но кто будет удалять?
}
int main() {
    int* data = createArray(10);
    // ... забыли delete[] data
    return 0;
}
```

- Функция возвращает указатель на динамическую память.
- Ответственность за освобождение перекладывается на вызывающую функцию.
- Легко забыть вызвать `delete[]`.

Двойное удаление

```
int* p = new int(5);
delete p;
delete p; // ОШИБКА! двойное удаление
```

- После первого `delete` память освобождена.
- Второй `delete` пытается освободить уже освобождённую память.
- Приводит к неопределённому поведению (UB).

Решение

После `delete` всегда обнуляйте указатель:

```
delete p;
p = nullptr;
// delete p; // теперь безопасно (delete nullptr ничего не делает)
```

Разыменование висячего указателя

```
int* p = new int(5);
delete p;
*p = 10; // ОШИБКА! обращение к освобождённой памяти
```

- После `delete` указатель всё ещё хранит адрес, но память уже не принадлежит программе.
- Разыменование такого указателя — неопределённое поведение.

Разыменование висячего указателя

```
int* p = new int(5);
delete p;
*p = 10; // ОШИБКА! обращение к освобождённой памяти
```

- После `delete` указатель всё ещё хранит адрес, но память уже не принадлежит программе.
- Разыменование такого указателя — неопределённое поведение.

После удаления нельзя обращаться к объекту. Указатель можно сбросить в `nullptr`, чтобы не хранить в нём старый адрес.

```
int* p = new int(5);
delete p;
p = nullptr;
// *p = 10; // UB: разыменование nullptr
```

Несоответствие new и delete

Ошибка 1: массив — обычный delete

```
int* arr = new int[10];
delete arr; // ОШИБКА! нужно delete[]
```

Ошибка 2: скаляр — delete[]

```
int* p = new int(5);
delete[] p; // ОШИБКА! нужно delete
```

- Для массивов используйте `delete[]`, для скалярных — `delete`.
- Несоответствие — неопределённое поведение.

Выход за границы

```
int* arr = new int[5];
arr[5] = 100; // ОШИБКА! индекс 5 вне диапазона (0..4)
delete[] arr;
```

- Компилятор не проверяет границы динамических массивов.
- Выход за границы может повредить другие данные или вызвать краш (HEAP CORRUPTION DETECTED).
- Это одна из самых распространённых ошибок в C++.

Утечка памяти при исключении

```
void f() {
    int* p = new int(5);
    // если здесь возникнет исключение, delete не вызовется
    throw std::runtime_error("error");
    delete p;
}
```

- Если между `new` и `delete` возникает исключение, память не освобождается.
- Это одна из причин, почему ручное управление памятью опасно.
- Решение: использовать умные указатели (будет позже).

Вопрос: что выведет этот код?

```
int* p = new int(5);
int* q = p;
delete p;
*q = 10;
cout << *q;
```

Вопрос: что выведет этот код?

```
int* p = new int(5);
int* q = p;
delete p;
*q = 10;
cout << *q;
```

Ответ: неопределённое поведение (UB). Может вывести 10, а может упасть.

Вопрос: что происходит в этом коде?

```
int* p = new int(5);
p = new int(10);
delete p;
```

Вопрос: что происходит в этом коде?

```
int* p = new int(5);
p = new int(10);
delete p;
```

Ответ: Утечка памяти для первого `int(5)`. Код удаляет только второе число, первое остаётся в памяти и теряется.

Выделение памяти под динамический одномерный массив

Синтаксис

```
int n = 10, n2 = 5;
int* arr = new int[n]; // массив из n элементов, элементы не инициализированы (содержат мусор)
int* arr2 = new int[n](); // все элементы инициализированы нулями
int* arr3 = new int[n2] {1, 2, 3, 4, 5}; // инициализация списком (C++11)
int* arr4 = new int[] {1, 2, 3, 4, 5};    // инициализация без указания размера (C++20)

delete[] arr;
delete[] arr2;
delete[] arr3;
delete[] arr4; // одним delete[] нельзя! Утечка!
```

- n может быть переменной — размер определяется во время выполнения.
- Массив хранится в куче.
- При успешном выделении памяти возвращается указатель на первый элемент. Если память выделить не удалось, бросается исключение `std::bad_alloc`.

Освобождение памяти

Правильный синтаксис

```
delete[] arr; // освобождение памяти, выделенной под массив
arr = nullptr; // обнуление указателя
```

- Всегда используйте `delete[]` для массивов.
- Не забывайте освобождать память, чтобы избежать утечек.
- Обнуление указателя после удаления — хорошая практика.

Доступ к элементам

```
int* arr = new int[10];
// через индексы
for (int i = 0; i < 10; ++i) {
    arr[i] = i * i;
}
// через указатели
for (int* p = arr; p < arr + 10; ++p) {
    cout << *p << " ";
}
delete[] arr;
```

- Доступ как к обычному массиву.
- Адресная арифметика работает так же, как со статическими массивами.

Передача динамического массива в функцию

```
void printArray(const int* arr, int size) {
    for (int i = 0; i < size; ++i)
        cout << arr[i] << " ";
}
int main() {
    int n = 5;
    int* data = new int[n]{1,2,3,4,5};
    printArray(data, n);
    delete[] data;
    return 0;
}
```

- Функция работает с указателем.
- Размер передаётся отдельно.
- const защищает массив от изменений.

Пример: сумма элементов

```
int sumArray(const int* arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; ++i)
        sum += arr[i];
    return sum;
}
int main() {
    int n;
    cin >> n;
    int* arr = new int[n];
    for (int i = 0; i < n; ++i) cin >> arr[i];
    cout << sumArray(arr, n);
    delete[] arr;
    return 0;
}
```

Работа с диапазонами

```
int sumRange(const int* begin, const int* end) {
    int sum = 0;
    for (const int* p = begin; p != end; ++p)
        sum += *p;
    return sum;
}
int main() {
    int* arr = new int[10]{1,2,3,4,5,6,7,8,9,10};
    cout << sumRange(arr, arr + 5);     // первые 5 элементов
    cout << sumRange(arr + 2, arr + 7); // элементы с индексами 2..6
    delete[] arr;
    return 0;
}
```

- Диапазон задаётся полуоткрытым интервалом [begin, end).
- end указывает на элемент за последним.

Изменение размера динамического массива

```
void resizeArray(int*& oldArr, int oldSize, int newSize) {
    int* newArr = new int[newSize]{}; // с инициализацией
    int copySize = (oldSize < newSize) ? oldSize : newSize;
    for (int i = 0; i < copySize; ++i)
        newArr[i] = oldArr[i];
    delete[] oldArr;
    oldArr = newArr;
}
```

- Нет способа просто изменить размер — нужно создать новый массив, скопировать в него данные, затем удалить старый массив и перебросить указатель.
- В случае изменения исходного массива — передавайте указатель на него по ссылке.
- Это дорогая операция — используйте её только при необходимости.

Изменение размера динамического массива (способ 2)

```
int* resizeArray(int* oldArr, int oldSize, int newSize) {
    int* newArr = new int[newSize]{}; // с инициализацией
    int copySize = (oldSize < newSize) ? oldSize : newSize;
    for (int i = 0; i < copySize; ++i)
        newArr[i] = oldArr[i];
    return newArr;
}
```

- Можно локально выделить память под новый массив, скопировать в него данные и вернуть указатель на новый массив.
- Плохо — забыть потом удалить этот новый локально созданный массив или старый массив в функции `main`.
- Плохо — если не знаете размер нового массива.

Вопрос: что выведет этот код?

```
int* arr = new int[3]{1,2,3};
int* p = arr;
cout << *p << " ";
p++;
cout << *p << " ";
delete[] arr;
```

Вопрос: что выведет этот код?

```
int* arr = new int[3]{1,2,3};
int* p = arr;
cout << *p << " ";
p++;
cout << *p << " ";
delete[] arr;
```

Ответ: 1 2. Указатель p успешно перемещается по массиву, память освобождается после вывода.

Вопрос: что выведет этот код?

```
int* arr = new int[3]{1,2,3};
delete[] arr;
cout << arr[0];
```

Вопрос: что выведет этот код?

```
int* arr = new int[3]{1,2,3};
delete[] arr;
cout << arr[0];
```

Ответ: неопределённое поведение (UB). Массив уже удалён, обращение к его элементу приводит к ошибке.

Пример: проверка симметричности

```
bool isSymmetric(const int* arr, int size) {
    if (size <= 1) return true;
    const int* left = arr;
    const int* right = arr + size - 1;
    while (left < right) {
        if (*left != *right) return false;
        left++;
        right--;
    }
    return true;
}
```

Копирование массива

```
int* copyArray(const int* source, int size) {
    int* dest = new int[size];
    for (int i = 0; i < size; ++i)
        dest[i] = source[i];
    return dest;
}
```

- Создаётся независимая копия данных.
- Изменения в копии не влияют на оригинал.

Вопрос: что выведет этот код?

```
int* a = new int[2]{1,2};
int* b = a;
delete[] a;
b[0] = 10;
cout << b[0];
```

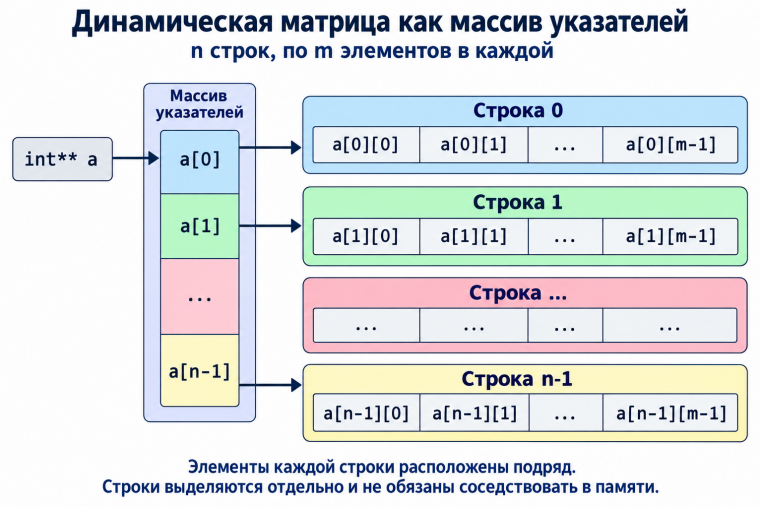
Вопрос: что выведет этот код?

```
int* a = new int[2]{1,2};
int* b = a;
delete[] a;
b[0] = 10;
cout << b[0];
```

Ответ: неопределённое поведение (UB). a и b указывают на одну память, которая уже удалена.

Двумерные динамические массивы (матрицы)

Массив указателей на строки. Каждая строка — отдельный динамический массив.



Выделение памяти под динамический двумерный массив (матрицу)

```
int rows = 3, cols = 4;
int** matrix = new int*[rows]; // массив указателей на строки
for (int i = 0; i < rows; ++i) {
    matrix[i] = new int[cols]; // каждая строка - массив
}
```

- Сначала выделяется массив указателей на строки.
- Затем для каждой строки выделяется память под столбцы.
- Доступ: `matrix[i][j]`.

Освобождение памяти

```
for (int i = 0; i < rows; ++i) {
    delete[] matrix[i]; // удаляем каждую строку
}
delete[] matrix; // удаляем массив указателей
matrix = nullptr;
```

- Освобождение в обратном порядке: сначала строки, затем массив указателей.
- Если забыть удалить строки — утечка памяти.

Передача матрицы в функцию

```
void printMatrix(int** matrix, int rows, int cols) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j)
            cout << matrix[i][j] << " ";
        cout << endl;
    }
}
```

Пример: создание и печать матрицы

```
int main() {
    int rows, cols;
    cin >> rows >> cols;
    int** mat = new int*[rows];
    for (int i = 0; i < rows; ++i) {
        mat[i] = new int[cols];
        for (int j = 0; j < cols; ++j)
            mat[i][j] = i * cols + j;
    }
    printMatrix(mat, rows, cols);
    for (int i = 0; i < rows; ++i) delete[] mat[i];
    delete[] mat;
    return 0;
}
```

Вопрос: что выведет этот код?

```
int** m = new int*[2];
m[0] = new int[2]{1,2};
m[1] = new int[2]{3,4};
cout << m[1][0];
// вычистка
```

Вопрос: что выведет этот код?

```
int** m = new int*[2];
m[0] = new int[2]{1,2};
m[1] = new int[2]{3,4};
cout << m[1][0];
// вычистка
```

Ответ: 3. `m[1][0]` — элемент второй строки (индекс 1) и первого столбца (индекс 0).

Ступенчатые двумерные динамические массивы

- Строки могут иметь разную длину.
- Выделение памяти для каждой строки отдельно.

```
int** mat = new int*[rows];
for (int i = 0; i < rows; ++i) {
    mat[i] = new int[i + 1]; // каждая строка длиннее предыдущей
}
```

- Экономия памяти, если строки разной длины.

Пример: верхнетреугольная матрица

```
int n = 5;
int** mat = new int*[n];
for (int i = 0; i < n; ++i) {
    mat[i] = new int[n - i];   // каждая строка короче предыдущей
    for (int j = 0; j < n - i; ++j)
        mat[i][j] = j + 1;
}
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ & 1 & 2 & 3 & 4 \\ & & 1 & 2 & 3 \\ & & & 1 & 2 \\ & & & & 1 \end{pmatrix}$$

- Каждая строка короче предыдущей и содержит `n - i` элементов.
- В коде память выделяется только под реально существующие элементы (на главной диагонали и выше неё).

Вопрос: общее количество заполненных элементов?

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ & 1 & 2 & 3 & 4 \\ & & 1 & 2 & 3 \\ & & & 1 & 2 \\ & & & & 1 \end{pmatrix}$$

Вопрос: общее количество заполненных элементов?

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ & 1 & 2 & 3 & 4 \\ & & 1 & 2 & 3 \\ & & & 1 & 2 \\ & & & & 1 \end{pmatrix}$$

Ответ: Общее количество заполненных элементов: `n*(n+1)/2` (сумма арифметической прогрессии от 1 до `n`).

Освобождение ступенчатой матрицы

```
for (int i = 0; i < rows; ++i) {
    delete[] mat[i];
}
delete[] mat;
```

- Та же последовательность, что и для обычной матрицы.

Вопрос: что выведет этот код?

```
int** m = new int*[2];
m[0] = new int[1]{1};
m[1] = new int[2]{2,3};
cout << m[1][1];
// вычистка
```

Вопрос: что выведет этот код?

```
int** m = new int*[2];
m[0] = new int[1]{1};
m[1] = new int[2]{2,3};
cout << m[1][1];
// вычистка
```

Ответ: 3. `m[1][1]` — второй элемент второй строки.

Проблемы ручного управления памятью

- Утечки памяти — легко забыть вызвать `delete`.
- Двойное удаление и висячие указатели.
- Исключения между `new` и `delete`.
- Сложность отслеживания времени жизни объектов.

Решение

В C++ существуют **умные указатели**, которые автоматически управляют временем жизни объектов.

Умные указатели (концепция)

- Умный указатель — объект, который ведёт себя как указатель, но автоматически (владеющие указатели) освобождает память, когда он больше не нужен.
- Основная идея: связать время жизни объекта с временем жизни умного указателя.
- В C++ есть три вида умных указателей:
  1. `std::unique_ptr` — единственный владелец (нельзя копировать).
  2. `std::shared_ptr` — совместное владение; объект уничтожается при исчезновении последнего владельца.
  3. `std::weak_ptr` — невладеющая (слабая) ссылка, которая не продлевает жизнь объекта (для предотвращения циклических ссылок).
- Подробно изучим в весеннем семестре.
- **Важно:** умные указатели — это стандартный способ управления памятью в современном C++.

Итоги лекции

- Познакомились с динамической памятью и областями памяти.
- Изучили операторы `new` и `delete`.
- Рассмотрели типичные ошибки: утечки, двойное удаление, висячие указатели.
- Научились создавать одномерные и двумерные динамические массивы.
- Узнали о ступенчатых массивах.
- Поняли, что ручное управление памятью — источник ошибок, и существует альтернатива — умные указатели.

Что дальше?

- Следующая лекция: `std::vector` — динамический массив из стандартной библиотеки.
- `vector` автоматически управляет памятью, что избавляет от многих проблем.
- На практике: закрепление работы с динамической памятью.

Вопросы?