Лекция 4 – Статические многомерные массивы, динамическое выделение памяти, динамические многомерные массивы

Воронин Андрей Андреевич

Кафедра прикладной математики и информатики

28 октября 2019 г.

Инициализация

Инициализация одномерного массива

```
const int DIM1 = 4;
int array[DIM1] = {0,1,2,3}
```

Инициализация двумерного массива

```
const int DIM1 = 3;
const int DIM2 = 5;

int array[DIM1][DIM2] = {
    { 1, 2, 3, 4, 5 },
    { 2, 4, 6, 8, 10 },
    { 3, 6, 9, 12, 15 }
};
```

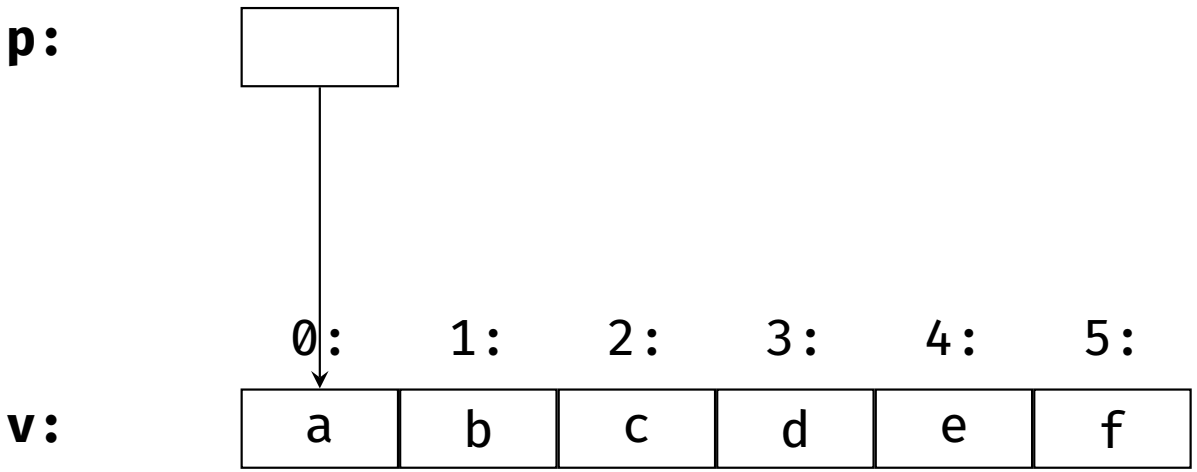
Ввод и вывод значений массива

```
#include <iostream>
using namespace std;

const int DIM1 = 3;
const int DIM2 = 5;

int array[DIM1][DIM2];

int main() {
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            array[i][j] = (i + 1) * 10 + (j +
            ↪  1);
        }
    }

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << array[i][j] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

```
11  12  13  14  15
21  22  23  24  25
31  32  33  34  35
```

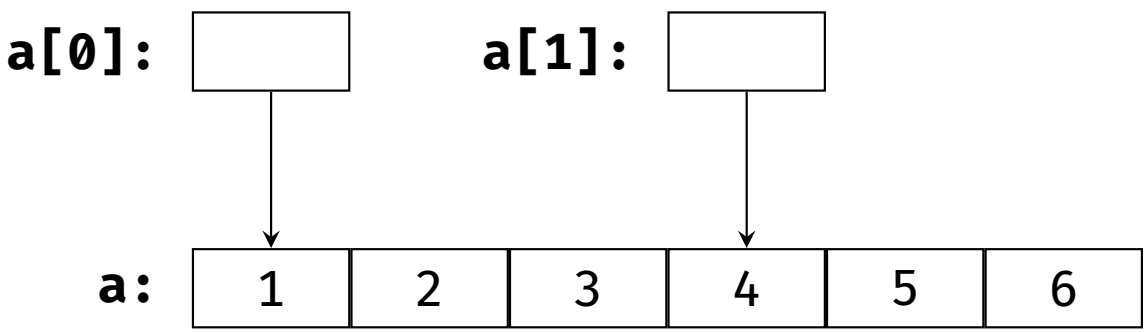
Расположение в памяти

```
#include <iostream>

int main()
{
    char v[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    char *p = &v[0];

    if (v == p)
        std::cout << "p == v";
    return 0;
}
```



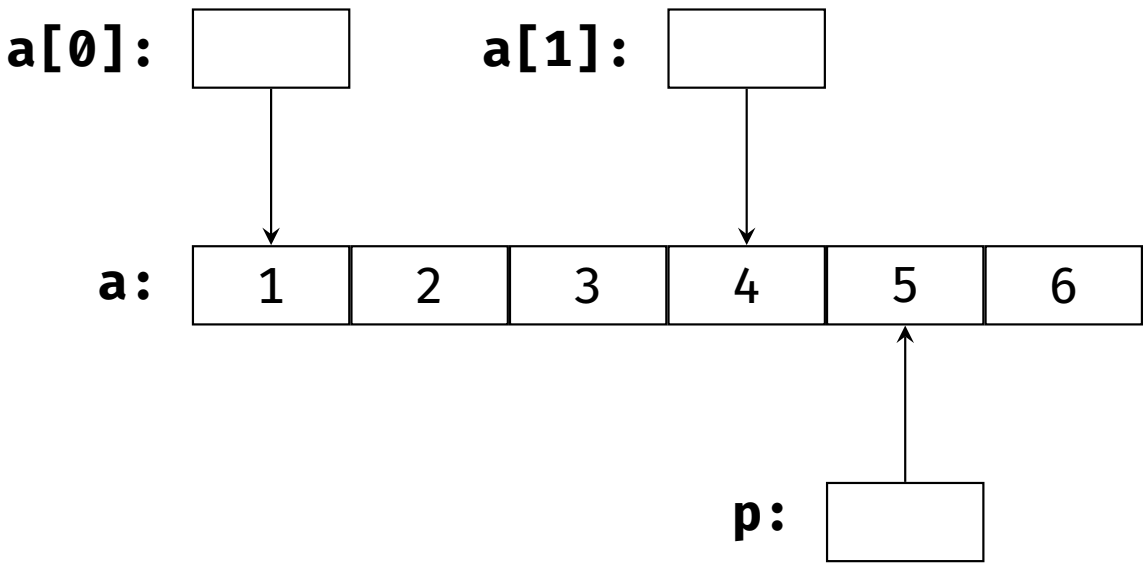
Расположение в памяти

```
int a[2][3] = {{1,2,3},{4,5,6}};

std::cout << a << std::endl;          //0x7ffdd91484e0
std::cout << a[0] << std::endl;       //0x7ffdd91484e0
std::cout << a[1] << std::endl;       //0x7ffdd91484ec
std::cout << &a[0][0] << std::endl; //0x7ffdd91484e0
std::cout << &a[1][0] << std::endl; //0x7ffdd91484ec
```

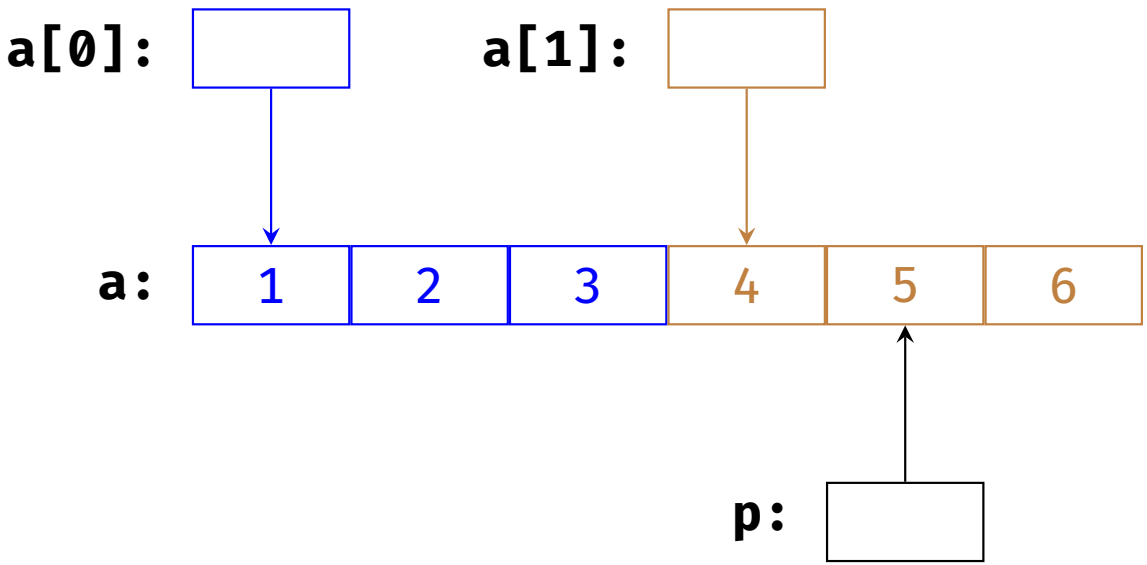


Расположение в памяти

```
int a[2][3] = {{1,2,3},{4,5,6}};

int v1 = a[1][1];
int v2 = *(*(a+1)+1);

int p = &a[1][1];
if (v1 == v2) std::cout << "v1 == v2";
```



Расположение в памяти



Адресная арифметика

```cpp
#include <iostream>
using namespace std;

const int DIM1 = 3;
const int DIM2 = 5;

int array[DIM1][DIM2];

int main() {
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            array[i][j] = (i + 1) * 10 + (j +
            ↪  1);
        }
    }

    int *ptr = (int *)array;    // так не надо
    ↪  делать ;-)

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << *(ptr + i * DIM2 + j) <<
            ↪  "\t";
        }
        cout << endl;
    }

    return 0;
```

```
11  12  13  14  15
21  22  23  24  25
31  32  33  34  35
```

Адресная арифметика

```cpp
#include <iostream>
using namespace std;

const int DIM1 = 3;
const int DIM2 = 5;

int array[DIM1][DIM2];

int main() {
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            array[i][j] = (i + 1) * 10 + (j +
            ↪  1);
        }
    }

    int *ptr = (int *)array;

    for (int i = 0; i < DIM1 * DIM2; i++) {
        cout << ptr[i] << "\t";
    }
    cout << endl;

    return 0;
}
```

```
11  12  13  14  15
↪  21  22  23  24
↪  25  31  32  33
↪  34  35
```

Двумерный массив как одномерный

```
const int DIM1 = 3;
const int DIM2 = 5;

int ary[DIM1 * DIM2];

int main() {

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            *(ary + i * DIM2 + j) = (i + 1) *
            ↪  10 + (j + 1);
        }
    }

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << *(ary + i * DIM2 + j) <<
            ↪  "\t";
        }
        cout << endl;
    }

    return 0;
}
```

```
11  12  13  14  15
↪  21  22  23  24
↪  25  31  32  33
↪  34  35
```

Необходимость динамического выделения памяти

Как статическое, так и автоматическое распределение памяти имеют два общих свойства:

- Размер переменной/массива должен быть известен во время компиляции.
- Выделение и освобождение памяти происходит автоматически (когда переменная создаётся/уничтожается).

Если нам нужно объявить размер всех переменных во время компиляции, то самое лучшее, что мы можем сделать — это попытаться угадать их максимальный размер, надеясь, что этого будет достаточно:

```
char name[30]; // будем надеяться, что пользователь
 ↪  введёт имя длиной менее 30 символов!
Monster monster[30]; // 30 монстров максимум
Polygon rendering[40000]; // этому 3d rendering-у
 ↪  лучше состоять из менее чем 40 000 полигонов!
```

Недостатки статического выделения памяти

- теряется память, если массив не используется, или используется не весь;
- статически выделяемая память выделяется из специального хранилища – **стека**; его размер как правило ограничен 1 МБ памяти, превышение данного значения ведет к завершению программы с ошибкой переполнения стека ”stack overflow”;
- теряется информация если ввод превосходит заранее заданный размер.

Оператор new

Определение

Динамическое выделение памяти — это способ запроса памяти из операционной системы запущенными программами по надобности. Эта память не выделяется из ограниченной памяти стека программы, а из гораздо большего хранилища, управляемого операционной системой — **кучи**. На современных компьютерах размер кучи может составлять гигабайты памяти.

```
int *ptr = new int; // динамически выделяем
 ↪  целочисленную переменную и присваиваем её адрес
 ↪  ptr, чтобы потом иметь доступ к ней
*ptr = 8; // присваиваем значение 8 только что
 ↪  выделенной памяти
```

Оператор **new** возвращает **указатель**, содержащий адрес выделенной памяти.

Оператор delete

```
// Предположим, что ptr ранее уже был выделен с
 ↪  помощью оператора new
delete ptr; // возвращаем память, на которую указывал
 ↪  ptr, обратно в операционную систему
ptr = 0; // делаем ptr нулевым указателем
 ↪  (используйте nullptr вместо 0 в C++11)
```

Хотя может показаться, что мы удаляем переменную, но это не так!
Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, и ей можно присвоить новое значение, как и любой другой переменной.
Обратите внимание, удаление указателя, не указывающего на динамически выделенную память, может привести к проблемам.

Стек и куча

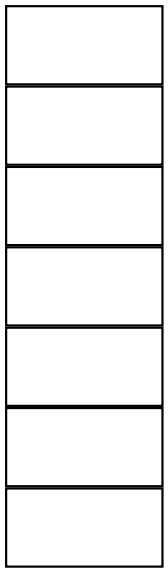


Рис.: stack

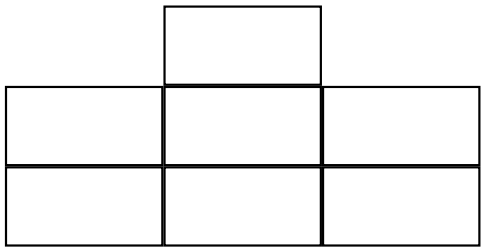


Рис.: heap

Висячие указатели

```
#include <iostream>

int main()
{
    int *ptr = new int; // динамически выделяем
    ↪  целочисленную переменную
    *ptr = 8; // помещаем значение в выделенную
    ↪  ячейку памяти

    delete ptr; // возвращаем память обратно в
    ↪  операционную систему. ptr теперь является
    ↪  висячим указателем

    std::cout << *ptr; // разыменование висячего
    ↪  указателя приведёт к неожиданным результатам
    delete ptr; // попытка освободить память снова
    ↪  приведёт к неожиданным результатам также

    return 0;
}
```

Висячие указатели

```
#include <iostream>

int main()
{
    int *ptr = new int; // динамически выделяем
    ↪  целочисленную переменную
    int *otherPtr = ptr; // otherPtr теперь указывает
    ↪  на ту же самую выделенную память, что и ptr

    delete ptr; // возвращаем память обратно в
    ↪  операционную систему. ptr и otherPtr теперь
    ↪  висячие указатели
    ptr = 0; // ptr теперь уже nullptr

    // Однако otherPtr по-прежнему является висячим
    ↪  указателем!

    return 0;
}
```

Нулевые указатели и динамическое выделение памяти

Нулевые указатели (указатели со значением 0 или nullptr) особенно полезны в процессе динамического выделения памяти. Их наличие как бы сообщаем нам: «Этому указателю не выделено никакой памяти». А это, в свою очередь, можно использовать для выполнения условного выделения памяти:

```
// Если ptr-у до сих пор не выделено памяти, то
 ↪  выделяем её
if (!ptr)
    ptr = new int;
```

Удаление нулевого указателя ни на что не влияет. Таким образом, в следующем нет необходимости:

```
if (ptr)
    delete ptr;
```

Вместо этого вы можете просто написать:

```
delete ptr;
```

Утечка памяти

Динамически выделенная память не имеет области видимости, т.е. она остаётся выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершит своё выполнение (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости обычных переменных. Например:

```
void doSomething(){
    int *ptr = new int;
}
```

Утечка памяти

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
int value = 7;
int *ptr = new int; // выделяем память
ptr = &value; // старый адрес утерян - произойдёт
 ↪  утечка памяти
```

Корректный способ освобождения памяти:

```
int value = 7;
int *ptr = new int; // выделяем память
delete ptr; // возвращаем память обратно в
 ↪  операционную систему
ptr = &value; // переприсваиваем указателю адрес
 ↪  value
```

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

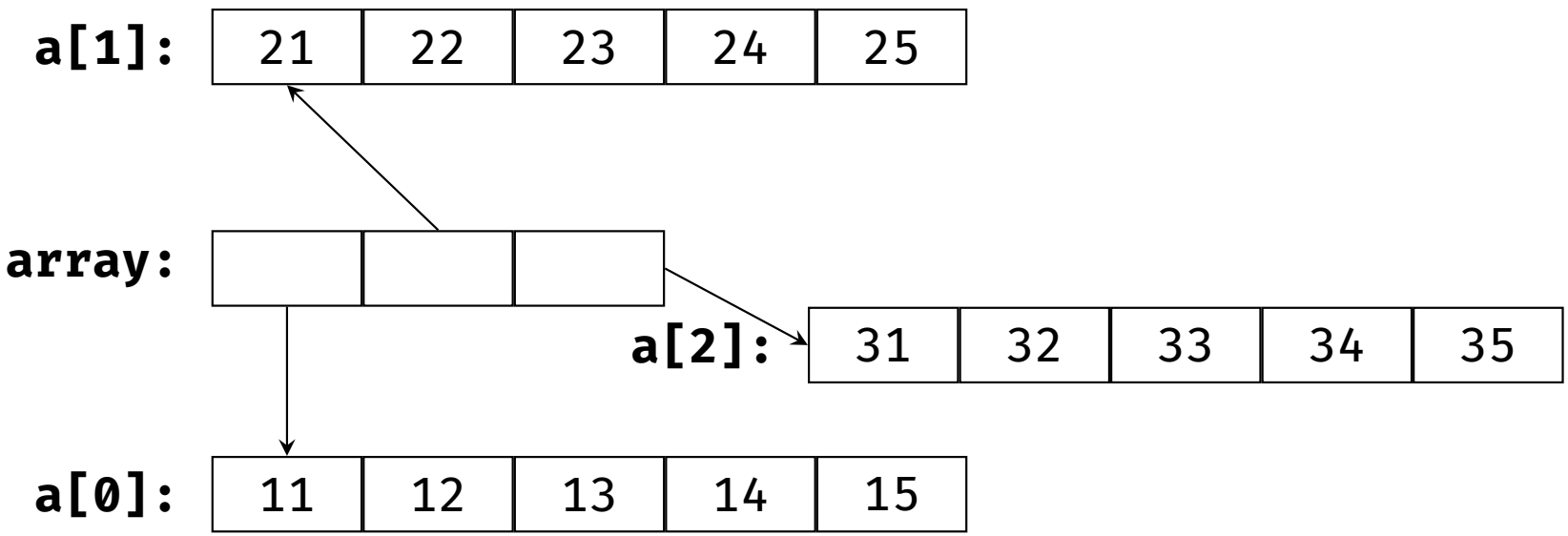
```
int *ptr = new int;
ptr = new int; // старый адрес утерян - произойдёт
 ↪  утечка памяти
```

Создание и уничтожение динамических многомерных массивов

```
const int DIM1 = 3;
const int DIM2 = 5;

int main() {
    int **array;    // (1)
    // создание
    array = new int * [DIM1]; // массив указателей (2)
    for (int i = 0; i < DIM1; i++) { // (3)
        array[i] = new int [DIM2]; // инициализация указателей
    }

    // работа с массивом
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            array[i][j] = (i + 1) * 10 + (j + 1);
        }
    }

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << array[i][j] << "\t";
        }
        cout << endl;
    }

    // уничтожение
    for (int i = 0; i < DIM1; i++) {
        delete [] array[i];
    }
    delete [] array;

    return 0;
}
```

(1) Для доступа к двумерному массиву объявляется переменная `array` типа указатель на указатель на тип (в данном случае это указатель на указатель на **`int`**).
(2) Переменная инициализируется оператором new, который выделяет память для массива указателей на **`int`**.
(3) В цикле каждый элемент массива указателей инициализируется оператором new, который выделяет память для массива типа **`int`**.
Освобождение памяти происходит строго в обратном порядке: сначала уничтожаются массивы значений типа **`int`**, а затем уничтожается массив указателей.
Работа с динамическим многомерным массивом синтаксически полностью совпадает с работой с многомерным С-массивом.

Расположение в памяти



Двумерный вектор

```
const int DIM1 = 3;
const int DIM2 = 5;
#include <vector>

using namespace std;

int main() {
    vector<vector<int>> v;

    // заполнение
    for (int i = 0; i < DIM1; i++){
        vector<int> temp;
        v.push_back(temp);
        for (int j = 0; j < DIM2; j++) {
            v.at(i).push_back(
                (i + 1) * 10 + (j + 1));
        }
    }

    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << v.at(i).at(j) << "\t"; // v[i][j]
        }
        cout << endl;
    }

    return 0;
}
```