Указатели, ссылки, структуры

Массивы

Ссылки C++

Структуры

Объединения

Статические массивы

Создание статического массива возможно, если размер может быть определен на стадии компиляции программы:

```
const int N = 8;
// константное выражение = может быть вычислено во время компиляции
constexpr int Square(int n) { return n * n; }

void withMatrices()
{
      int a1[1];
      int a2[N];
      int a3['a’];
      int a4[Square(2)];
..
}
```

Размер массива

- При работе со **статическими** массивами размер массива можно узнать, используя оператор sizeof или функцию std::size

```
long m[12];
cout << "Размер массива в байтах " << sizeof(m) << endl;
cout << "Количество элементов массива " << std::size(m) << endl;
```

```
Размер массива в байтах 48
Количество элементов массива 12
```

- В случае многомерных массивов std::size выдает количество элементов по одному измерению

```
long long mm[2][5];
cout << "Размер массива m[2][5] в байтах " << sizeof(mm) << endl;
cout << "Количество строк " << size(mm) << endl;
cout << "Количество столбцов " << size(mm[0]) << en
```

```
Размер массива m[2][5] в байтах 80
Количество строк 2
Количество столбцов 5
```

array-to-pointer-decay

- array-to-pointer-decay = *понижение типа (сведение)*
- (Статический) массив не является аналогом указателя на элементы
- Переход от массива к указателю предполагает потерю информации о размере массива
- Такой переход выполняется неявно при передаче массива в функцию, в этом случае размер нужно передавать отдельно

```
// приведенные сигнатуры функций идентичны
void showElements(int m[4], int size);
void showElements(int m[],  int size);
void showElements(int *m,   int size);
```

- Приведенные сигнатуры функций идентичны; аргумент интерпретируется как `int *` и передача осуществляется адреса на область памяти, содержащую элементы массива (указателя)

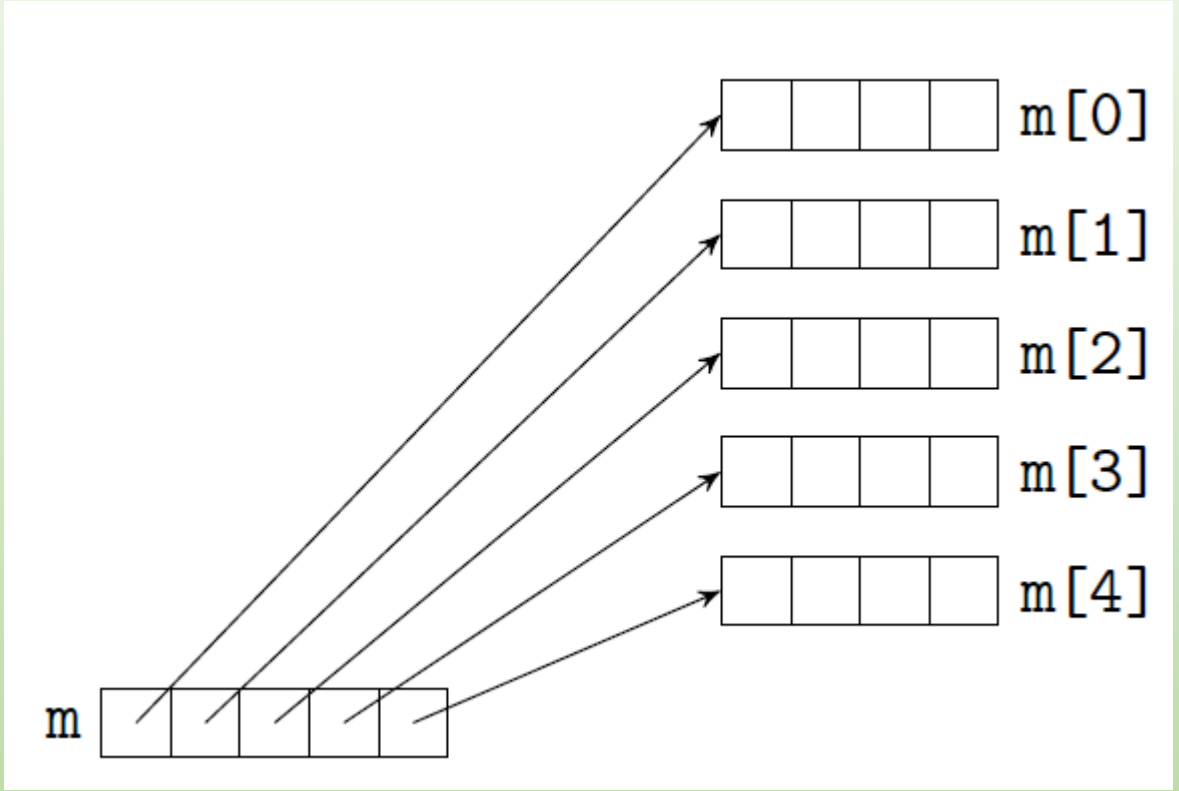
Указатель на массив ≠ указатель на первый элемент

```
int a[10];
// указатель на первый элемент массива
int* p = a;
// указатель на массив
int(*q)[10];
q = &a;
cout << size(*q) << endl;
```

- Указатель на массив вводится с указанием размера массива; размер массива доступен через `std::size`.

Многомерные динамические массивы

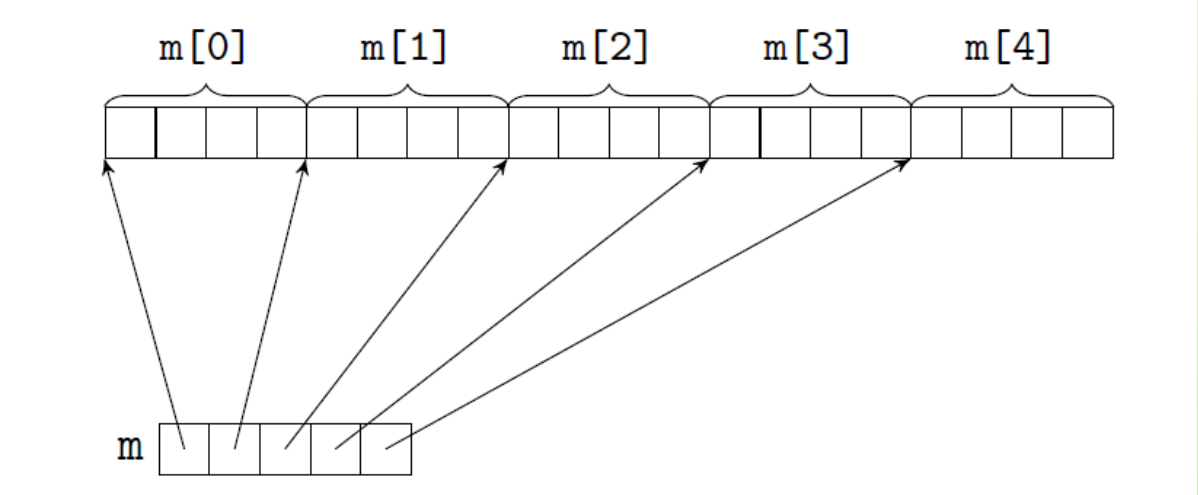
```
int** m, ** b, ** c;
// m - это указатель на массив указателей
m = new int* [N];
int s = 0;
for (int i = 0; i < N; i++)
{
        // m[i] - указатель на множество
        // целых чисел (строка)
      m[i] = new int[N];
      for (int j = 0; j < N; j++)
      {
            m[i][j] = ++s;
      }
}
```



Оптимизация хранения данных

```
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;

delete [] m [0];
delete [] m;
```




```
// храним все данные в одномерном массиве
int * m = new int  [N * M];
for ( size_t i=0; i != N; ++i)
    for ( size_t j=0; j != M; ++j)
       m[i * M + j] = rand();
```

Указатели и Си-строки

- Для работы со строками в Си применяются типы char*
- С-строка заканчивается нулевым символом ‘\0’ (в числовом представлении 0).

```
// длина С-строки
int strlen(char* s)
{
    int i=0;
    for (; *s++; i++);
    return i;
}
```

```
char* revers(char* s)
{
  int i, j, n = strlen(s);
  char ch, * p, * q;
  for (p = s, q = s + (n - 1); p < q; p++, q--)
  ch = *p; *p = *q; *q = ch;
  return s;
}
```

```
void strcpy(char* out, char* in)
{
    while (*out++ == *in++);
}
```

Применение указателей в коде

- Для работы с динамической памятью

```
Point* q = new Point;
```

- Для работы с массивами
- для возможности изменить значение переменной внутри функции

```
void getDoubled(double* v) { *v= *v * *v; }
```

- для возможности «вернуть» из функции несколько значений

```
void getMinMax(double values[], double* min, double* max);
```

- для «экономной» передачи данных в функцию

```
Person * whoIsOlder(Person * p1, Person * p2);
```

Ссылочный тип данных

- Для передачи функций в качестве аргументов

```
// double transform(double value, double(*f)(double)) { return f(value); }
double d = transform(3.14, sqrt);
```

Ссылки

- Ссылки появились в C++ для упрощения синтаксиса работы с указателями
- Ссылки часто интерпретируют как псевдонимы переменных
- Ограничения ссылок:
  - ссылочная переменная инициализируется при объявлении
  - ссылочную переменную нельзя переопределять
  - не поддерживается адресная арифметика
  - ссылка должна связываться с конкретным «местом хранения»
- Механизм работы ссылок не определяется на уровне стандарта, а определяется компилятором
- Наиболее часто ссылки используются как аргументы функций и возвращаемое значение.

Ссылочные типы

```
int n = 8;
// указатель
int* p = &n;
*p = 34;
// ссылочный тип = псевдоним
int& r = n;
r = 17;
```

n = 34 <000000FA35AFF8F0>, p = 000000FA35AFF8F0, &p =000000FA35AFF8F8
n = 17 <000000FA35AFF8F0>, &r = 000000FA35AFF8F0

Применение указателей - swap1

```
// Функция работает с копиями значений, а не с самими ячейками памяти.
void swap1(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 5, b = 7;
    swap1(a, b);
    cout << "a = " << a << ", b = " << b << endl;
}
```

Применение указателей – swap2

```
// Функция работает с адресами ячеек памяти.
void swap2(int* px, int* py)
{
    int t = *px;
    *px = *py;
    *py = t;
}
int main()
{
    int a = 5, b = 7;
    swap2(&a, &b);
    cout << "a = " << a << ", b = " << b << endl;
}
```

- Указатель используется для возможности изменить значение
- Адрес переменной сам по себе не важен
- По указателю происходит чтение значения

Применение ссылок – swap3

```
// Функция работает с адресами ячеек памяти.
void swap3(int& px, int& py)
{
    int t = px;
    px = py;
    py = t;
}
int main()
{
    int a = 5, b = 7;
    swap3(a, b);
    cout << "a = " << a << ", b = " << b << endl;
}
```

Код стал «чище»
Нет операций «разыменования» для доступа к значению
Нет операции взятия адреса

Левые и правые (lvalue, rvalue)

```
// передача по ссылке
void Change(int& value)
{      value += 10;    }
void main()
{
       int x = 0;
       change(x);
       change(10);          // Ошибка компиляции!
       change(x + 5);       // Ошибка компиляции!
       int& ref_x = x + 3; // Ошибка компиляции!

       just_read(x + 5);  // Работает с  void just_read(const int& value)
}
```

lvalue - переменная, элемент массива, разыменованный указатель, член структуры

const

- Ключевое слово const вводит дополнительную защиту на стадии компиляции от изменением

```
int x = 23;
const int* p1 = &x; // указатель на константу
int const* p2 = &x; // указатель на константу
*p1 = 44;     // ошибка
p2 = 0;       // OK

int* const p3 = &x; // константный указатель
*p3 = 40; // OK
p3 = 0; // ошибка

// константный указатель на константу
int const* const p4 = &x;
*p4 = 40; // ошибка
p4 = 0; // ошибка 3
```

const с ссылками

```
int a = 10;
int& const b = a; // ошибка
int const& c = a; // ссылка на константу
```

- Ссылки на константы часто используются как аргументы функций для «экономной» передачи данных

```
void justReadBigValue(BigValue const& r);
```

Составные типы: структуры (классы)

- Структуры (как и классы) позволяют сгруппировать логически связанные данные

```
struct Point {
        double x;
        double y;
};
```

- Структуры объявляются, как правило, вне определения функций (преимущественно в заголовочных файлах).
- В коде имя структуры используется аналогично встроенному типу данных

```
Point p;
Point* q = new Point;
// размер структуры
auto bytes = sizeof(p);
```

Структуры: доступ к полям

- Доступ к отдельным полям структуры обеспечивается с помощью оператора «.»

```
double distance(Point p1, Point p2)
{
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return sqrt(dx * dx + dy * dy);
}
```

- При работе с экземпляром структуры через указатель доступ к полям обеспечивается через оператор -> (*селектор выбора*)

```
Point* q = new Point;
q->x = 23;
q->y = 73;
```

Инициализация структур

- При определении структурной переменной возможна инициализация её полей:

```
struct Person {
        string fullName;
        int birthYear;
        string country;
};
// Определяем переменные внутри функции
Person cppCreator = { "Bjarne Stroustrup", 1950, "Denmark" };
Person cCreator1 { "Dennis MacAlistair Ritchie", 1941, "Usa" };
Person cCreator2 { "Kenneth Thompson" };
// Размещаем экземпляр в динамической памяти
string sUsa = "Usa";
auto cppAuthor = new Person { "Herbert Schildt", 1951, sUsa };
```

```
// Доступно с C++ 20 (designated initializer)
Person anonyme { .birthYear = 2020 };
```

Копирование структурных переменных

- Прямое присвоение структурных переменных приводит к копированию области памяти:

```
Person cppCreatorCopy = cppCreator;
```

- Возможно введение «посредников» для доступа к одной и той же области памяти:

```
// Указатель на область памяти cppCreator
Person* cppCreatorPointer = &cppCreator;
// Ссылка на cppCreator
Person& cppCreatorReference = cppCreator;
```

cppCreator:	**0000000D7C9EFD60**
cppCreatorCopy:	**0000000D7C9EFDB0**
cppCreatorPointer:	**0000000D7C9EFD60**
cppCreatorReference:	**0000000D7C9EFD60**

Хранение структур

- Хранение полей структуры в памяти организовано последовательно

```
struct Point { float x; float y; } p;
// размер структуры
auto bytes = sizeof(p);
// адреса полей
auto addr1 = &p.x, addr2 = &p.y;
```

sizeof(p) = 8
&p.x = 00000025142FF890,
&p.y = 00000025142FF894

Объединения

Unions are used when you want to save space by reusing space for information one type at a time. I don't use them much.

- В С вводится структурный тип «объединение» для работы с одной областью памяти через «призму» разных типов (все поля начинаются с одного адреса)

```
union Flag {
        int all;
        char f[4];
} flag;
flag.f[0] = 1;
flag.f[1] = 2;
Flag.f[2] = 3;
flag.f[3] = 4;
cout << "flag = " << hex << flag.all << endl;
cout << "sizeof(flag) = " << sizeof(flag) << endl;
cout << &flag.all << ", " << &flag.f << endl;
```

```
flag = 4030201
sizeof(flag) = 4
0000006D5319F5A0, 0000006D5319F5A0
```