

ОснОвы Программирования

Лекция #4

Темы:

1. Ссылки
2. Динамическая память
3. Многомерные массивы
4. Константные указатели

Недостатки указателей

- Использование указателей синтаксически загромождает 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).

Ссылки

- Для того, чтобы исправить некоторые недостатки указателей, в C++ введены ссылки.

```
void swap(int &a, int &b)
{
    int t = b;
    b = a;
    a = t;
}
int main()
{
    int k = 10, m = 20;
    swap(k, m);
    cout << k << ' ' << m << endl; // 20 10
    return 0;
}
```

Различия ссылок и указателей

- Ссылка не может быть неинициализированной.

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

- У ссылки нет нулевого значения.

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

- Ссылку нельзя переинициализировать.

```
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
```

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 *p1 = &l; // p1 указывает на переменную a  
int &&l1 = l; // ошибка
```

- Нельзя создавать массивы ссылок.

```
int *mp[10] = {}; // массив указателей на int  
int &m1[10] = {}; // ошибка
```

- Для ссылок нет арифметики.

lvalue и rvalue

- Выражения в C++ можно разделить на два типа:
 1. lvalue — выражения, значения которых являются ссылкой на переменную/элемент массива, а значит могут быть указаны слева от оператора =
 2. rvalue — выражения, значения которых являются временными и не соответствуют никакой переменной/элементу массива
- Указатели и 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
```

Время жизни переменной

```
int *foo()  
{  
    int a = 10;  
    return &a;  
}  
int &bar()  
{  
    int b = 20;  
    return b;  
}  
  
//.....  
  
int *p = foo();  
int &l = bar();
```


Динамическая память

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

```
// Не уместится на стек  
double m[10000000] = {}; // 80 Mb
```

- Время жизни локальных переменных ограничено временем работы функции.
- Динамическая память выделяется в сегменте данных.
- Структура, отвечающая за выделение дополнительной памяти, называется кучей (heap) (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.

Выделение памяти 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** 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

```
// 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 значений типа int
m = new int[1000];
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.
- Утечка 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
```

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 (или nullptr)  
free(q2); // правильно работает для q2 = 0
```

Многомерные массивы

- Двумерный массив – объект данных T $a[N][M]$, который:
 - Содержит N последовательно расположенных в памяти строк по M элементов типа T в каждой;
 - В общем и целом инициализируется аналогично одномерным массивам;
 - По характеристикам выравнивания идентичен объекту T $a[N * M]$, что сводит его двумерный характер к удобному умозрительному приему, упрощающему обсуждение и визуализацию порядка размещения данных
- Массивы размерности больше двух считаются **многомерными**, при этом $(N + 1)$ -мерные массивы индуктивно определяются как линеаризованные массивы N -мерных массивов, для которых справедливо все сказанное об одно- и двумерных массивах

Многомерные массивы

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};  
int b[2][3] = {{0, 1}, {2, 3, 4}};  
int c[2][3] = {0, 1, 2, 3, 4};
```

//результаты:

a: {1, 2, 3, 4, 5, 6}

b: {0, 1, 0, 2, 3, 4}

c: {0, 1, 2, 3, 4, 0}

// определение массивов размерности больше 2

```
double d[3][5][10];
```

Многомерные массивы и указатели

- Для многомерных массивов справедлив ряд тождеств, отражающих эквивалентность соответствующих выражений языка C. Так для двумерного массива T $a[N][M]$ справедливо:
 - $a == \&a[0]; a + 1 == \&a[i];$
 - $*a = a[0] == \&[0][0];$
 - $**a == *\&a[0][0] == a[0][0];$
 - $a[i][j] == (*(a + i) + j);$
- Использование операции разыменования $*$ не имеет каких-либо преимуществ перед доступом по индексу, и наоборот.

Многомерные массивы и указатели

// указатели на массивы и массивы указателей

```
int k[3][5];
```

```
int (*pk)[5]; // указатель на массив int[5]
```

```
int *p[5]; // массив указателей (int*)[5]
```

// примеры использования (все – допустимы)

```
pk          = k; // аналогично: pk = &k[0][0];
```

```
pk[0][0]    = 1; // аналогично: k[0][0] = 1;
```

```
*pk[0]      = 2; // аналогично: k[0][0] = 2;
```

```
**pk        = 3; // аналогично: k[0][0] = 3;
```

Совместимость указателей: пример

// определения

```
double *pd, **ppd;
```

```
double (*pda)[2];
```

```
double dbl32[3][2];
```

```
double dbl23[2][3];
```

// допустимые примеры использования

```
pd = &dbl32[0][0];           // double* -> double*
```

```
pd = dbl32[1];               // double[] -> double*
```

```
pda = dbl32;                 // double(*)[2] -> double(*)[2];
```

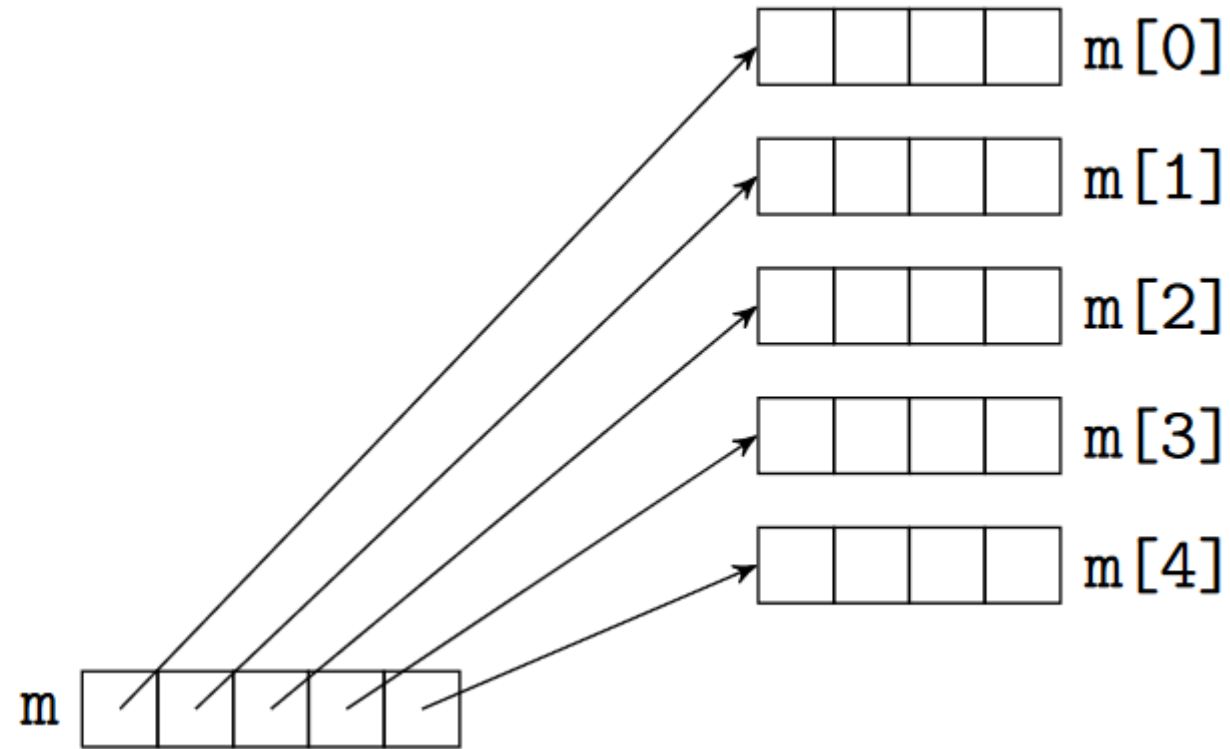
```
ppd = &pd;                   // double** -> double**
```

// недопустимые примеры использования

```
pd = dbl32;                  // double[][] -> double*
```

```
pda = dbl23;                 // double(*)[3] -> double(*)[2]
```

Двумерный массив



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

Модификатор const

```
const int p = 4;
```

```
p = 5; // ошибка
```

Указатели на константы и константные указатели

- Различное положение квалификатора `const` в определении указателя позволяет вводить в исходном коде программ 4 разновидности указателей:
 1. “обычный” указатель – изменяемый указатель на изменяемую через него область памяти;
 2. Указатель на константу – изменяемый указатель на неизменяемую область памяти;
 3. Константный указатель – неизменяемый указатель на изменяемую через него область памяти;
 4. Константный указатель на константу – неизменяемый указатель на неизменяемую через него область памяти.

Указатели на константы и константные указатели

// пример 1: определение

int* p1;	// обычный указатель
const int* pc2;	// указатель на константу
int* const cp3;	// константный указатель
const int* const cpc4;	// константный указатель на константу

// пример 2: совместимость T*, const T* и const T**

int* pi;	
const int* pci;	
const int** ppci;	
pci = pi;	// допустимо: int* -> const int*
pi = pci;	// недопустимо: const int* -> int*
ppci = &pci;	// недопустимо: int** -> const int**

Указатели на константы и константные указатели

// можно использовать со ссылками:

```
int p = 4;
```

```
const int& x = p; // нельзя через x поменять значение p;
```

```
x = 5; // ошибка
```

// константная ссылка – нонсенс

```
int& const x; // не имеет смысла
```

```
int* const cpi = &i;
```

// идентичен

```
int& ri = i;
```

Указатели на константы и константные указатели

// гарантия, что переданный параметр не будет изменен

```
void f1(const string& s);
```

```
void f2(const string* sptr);
```

```
void f3(string s); // будет работа с локальной копией
```