# Типы в C++

Базовые типы в C++

Массивы в  C++

Указатели и ссылки C++

# Встроенные типы

Конкретный размер типа зависит от компилятора и процессора, не определяется на уровне стандарта

Целочисленные типы:

- Тип `short` – не меньше 16 битов
- Тип `int` – не меньше `short`
- Тип `long` – не меньше 32 и не меньше `int`
- Тип `long long` – не меньше 64 и не меньше `long`

С каждым целочисленным типом можно использовать модификатор `unsigned`. В этом случае вдвое увеличивается множество положительных чисел

# Целочисленные типы

```
int n_int = INT_MAX;          // инициализация переменных максимальными значениями
short n_short = SHRT_MAX;     // константы определены в файле climits
long n_long = LONG_MAX;
long long n_llong = LLONG_MAX;

cout << "int is " << sizeof (int) << " bytes." << endl;
cout << "short is " << sizeof n_short << " bytes." << endl;
cout << "long is " << sizeof n_long << " bytes." << endl;
cout << "long long is " << sizeof n_llong << " bytes." << endl;

cout << "Maximum values:" << endl;
cout << "int: " << n_int << endl;
cout << "short: " << n_short << endl;
```

# Типы с плавающей точкой

Типы с плавающей точкой:

- Тип `float` - 4 байта, 7 значащих цифр.
- Тип `double` - 8 байт, 15 значащих цифр
- Тип `long double` – не меньше, чем double

# Символьные типы

- Тип `char` - 1 байт;
- В стандарте не постулируется является ли char знаковым или нет, допускаются явные определения:

```
unsigned char uc = 0;
signed char sc = -100;
```

- Тип `wchar_t` – 2 байта для работы с Unicode

- Начиная с C++ 11, доступны типы `char8_t, char16_t, char32_t` для работы со строками и символами

# Преобразование типов

- Автоматически выполняются преобразования арифметических типов в случае разных типов в выражении
  - при инициализации

```
long long l = 5;
```

  - При комбинировании в выражении

```
int x = 1;
int y = 3;
int result = x / y;
```

  - При передаче аргументов

```
int max(int a, int b) { return (a > b ? a : b); }
int main() {
      float f = max(3.14, 5.34);
      ..
}
```

# Преобразование типов

```
float f = 5;      // int -> float
// C++ проводит неявные сужающие преобразования без ошибок компиляции
// float -> int, потеря точности
int n = 8.9832;
// float -> int, неопределенный результат
int x = 7.2E12;
```

```
f = 5.000000
n = 8
x = 1634811904
```

# Инициализация переменных

*“В зависимости от вашей точки зрения выбор синтаксиса для инициализации объектов в C++ 1 1 либо очень богатый, либо запутанный и беспорядочный.”* (Скотт Мейерс, Эффективный и современный C++)

```
// традиционный синтаксис от Си
int x = 1;
int y(2);
// универсальный синтаксис инициализации C++11
// (списковая инициализация или braced initialization)
int z = { 3 };
int q { 4 };
// !Ошибка
// !Запрет на неявные сужающие преобразования при {}-init
int q { 3.14 };
```

# auto

- В C++ 11 появился механизм автоматического вывода типов

```
auto i = 34;                          // i - int
auto f = 45.12f;                  // f - float
auto l = 324234L;                 // l - long
auto ls = L"Hello, world";            // const wchar_t *
auto cs = "Hello, world";             // const char*
auto a = f1(2);
```

- Основное назначение auto при работе со сложными типами

```
map<string, int> dictionary{
        {"uno", 1}, {"dos", 2}, {"tres", 3}, {"cuatro", 4}, {"cinco", 5}
};
for (auto item : dictionary)
        cout << item.first << ": " << item.second << endl;
```

# Указатели

- C/C++ поддерживает непосредственную работу с адресами ячеек памяти

```
int n = 5;
// Оператор & позволяет получить адрес любой переменной
cout << "Address of n = " << &n << endl;
```

```
Address of n = 000000656477F7B0
```

- Для работы с адресами ячеек памяти в C/C++ вводятся *указатели*
- Указатель – переменная, хранящая адрес ячейки памяти
- Как правило, применяются *типизированные указатели*;
- Объявление указателя с помощью символа *:

```
int n = 5;
// «устанавливаем» указатель p на переменную n
int* p = &n;
```

# Указатели

- Оператор * (*разыменования*) позволяет получить значение по указателю :

```
int b = 5;
int* p = &b;
int a = *p;
```

- Указатели предоставляют прямой доступ к ячейкам памяти и опосредованный доступ к переменным
- Указатели используются как аргументы функций для возможности изменять значения
- Указатели используются при работе с последовательными блоками памяти (*массивы*)

# Указатели ≠ числа

- Указатели не являются числами

```
int* p = 455;
```

```
int* p = (int*) 455;
```

- Допускается создание «нулевого» указателя

```
int* p = 0;
```

- В современном C++ рекомендуется использовать

```
int* p = nullptr;
```

- Нулевые указатели используются как индикаторы «ошибок» в функциях и для устранения проблемы удаления неинициализированного указателя.

# void* или безтиповый указатель

- Можно вводить и безтиповые указатели, но операция разыменования потребует уточнения типа указателя:

```
int* p = &n;
int a = *p;
void* v = p;
// Печатаем адрес указателя p
cout << "p = " << p << endl;
// Читаем значение по указателю p
cout << "*p = " << *p << endl;
cout << "v = " << v << endl;
cout << "*v=" << *(int*)v << endl;
```

# 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;
}
```

# Массивы

- Массив – набор однотипных элементов, расположенных в памяти друг за другом (*последовательно*)
- В C++ поддерживаются массивы как в статической памяти (*стэковой*), так и в динамической памяти (heap)

```
// совмещение определения массива с инициализацией
int a[5] = { 1, 2, 3, 4, 5 };
// размер определяется компилятором
int b[] = { 1, 2, 3 };
// заполнение нулями с 3-го по 5-ый
int c[5] = { 10, 5 };
 // многомерный массив;
int m[500][500];
```

# p++;

- «Адресная арифметика»: операции +, -, ++, --, +=, -= применительно к указателям позволяют смещаться по памяти
- Шаг смещения определяется типом указателя:

```
short x = 6;
short y = 7;
short z = 8;
short* q = &x;
cout << "*q= " << *q << "(" << q << ")" << endl;
q++; // сдвигаемся по памяти вперёд на 2 байта
cout << "*q= " << *q << "(" << q << ")" << endl;
```

```
*q = 6(00AFF824)
*q = -13108(00AFF826)
```

# Массивы и указатели

- Адресная арифметика при работе с массивами позволяет *проходить* по элементам, так как для элементы массива расположены последовательно в памяти:

```
int m[5] = { 1, 2, 3, 4, 5 };
int* p = &m[0];
cout << "*p = " << *p << " (" << p << ")" << endl;
p++; // смещаемся на один элемент вперёд
cout << "*p = " << *p << " (" << p << ")" << endl;
p += 3; // смещаемся ещё на три элемента вперёд
cout << "*p = " << *p << " (" << p << ")" << endl;
```

```
*p = 1 (012FFC94)
*p = 2 (012FFC98)
*p = 5 (012FFCA4)
```

# Массивы и указатели

- Имя массива = указатель на первый элемент

```
int m[5] = { 1, 2, 3, 4, 5 };
// q указывает на элемент #0
int* q = m;
for (; q < m + 5; q++)
{
    cout << "item " << (q - m); // вычисляем номер элемента
    cout << " - value " << *q << endl;
}
```

# *(p+2) == p[2]

- При работе с указателем можно использовать [ ] для смещения
- Передача массивов в функцию осуществляется как правило через указатели

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

# Динамическая память

- Для хранения данных можно использовать статическую (стэковую) память или динамическую («куча», heap)
- Стэковая память проще для манипуляций, более производительна. Но вся информация о блоке памяти (тип элементов, количество элементов) должна быть известна на стадии компиляции (до запуска программы)
- Размер стэковой памяти гораздо меньше чем динамическая память, поэтому для хранения больших массивов данных она не предназначена.
- Указатели позволяют задействовать динамическую память для хранения данных
- В Си для управления динамической памятью используются функции `malloc(), calloc(), free()`
- В C++ рекомендуется использовать `new/delete`

# Управление динамической памятью

- Управление динамической памятью (выделение и освобождение) осуществляется вручную:

```
// храним в статической памяти
int x = 7;
// выделяем место в динамической памяти
int* p = new int;
*p = 10;
// освобождаем память
delete p;
```

- Оператор `delete` приводит к ошибке на неинициализированных указателях в отличие от `nullptr`
- Повторное освобождение памяти приводит к ошибке

# Динамическая память: массивы

- Для скалярный значений и массивов используются разные версии операторов: либо `new/delete`, либо `new T[]/delete[]`
- Для работы с массивами нужно указать размер выделяемой памяти (количество элементов)

```
int N = 200000;
int* q = new int[N];
q[0] = 0;
q[1] = 2;
q[2] = 3;
delete[] q;
```

# Типовые ошибки с указателями

- Объявление нескольких указателей в одной строке

```
int* s, t;
```

- Запись значений по неинициализированным указателям

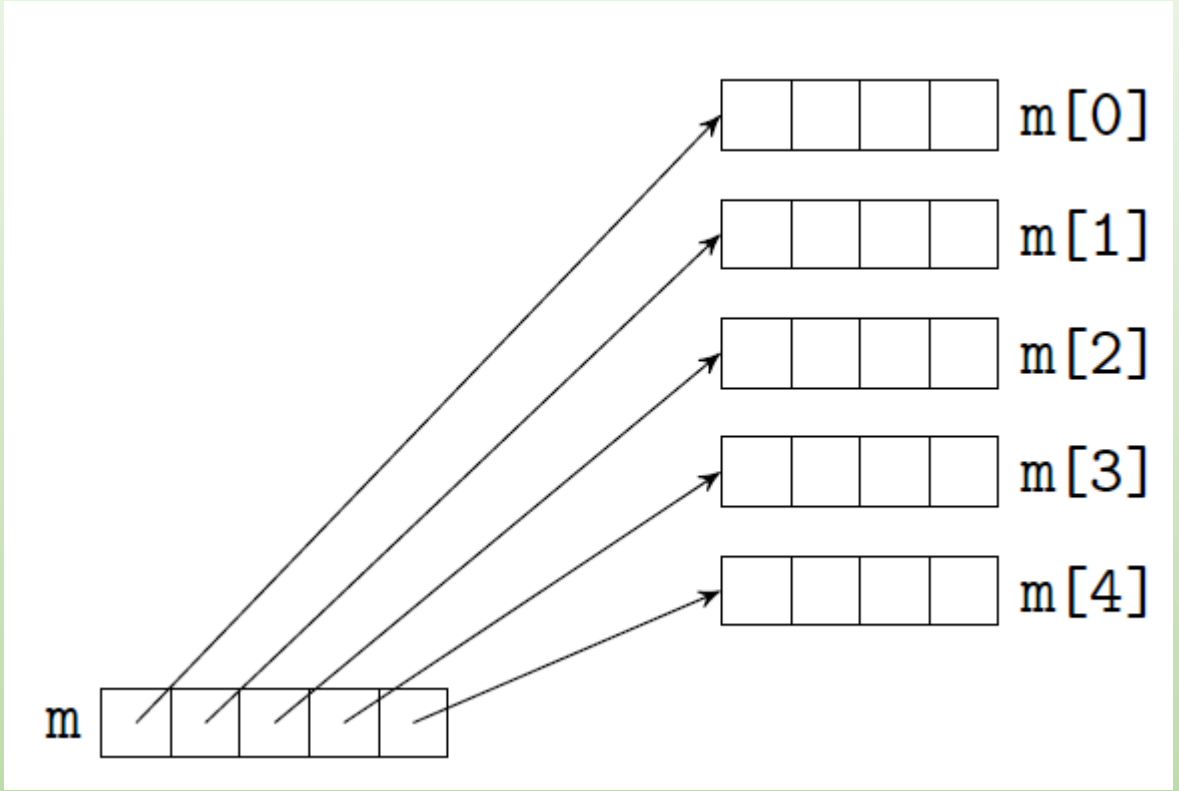
```
int *w, *v;
*w = -1; *v = 234;
```

- Подвешенные указатели (dangling pointers)

```
int* s = new int;
int* t = new int;
*s = 234;
*t = 123;
s = t;
```

# 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;
      }
}
```



# Освобождение памяти

```
// Освобождение памяти
for (int i = 0; i < N; i++)
{
	// удаляем каждую строку
	delete[] a[i];
}
delete[] a;
```