Курс лекций "Программирование"
Основы программирования на языках C и C++

Лекция 5. Массивы, строки, указатели

Глухих Михаил Игоревич, к.т.н., доц.
mailto: glukhikh@mail.ru

Что такое массив?

- Набор однотипных данных, расположенных в памяти последовательно
- Массивы используются для хранения таблиц, матриц, последовательностей и так далее
- Элемент массива – одна из ячеек
- Размерность массива – количество элементов в массиве
- Индекс (в массиве) – номер конкретного элемента в массиве

Определение массива в C/C++

```
// Без инициализации
// Массив из 20 целых чисел, заполняется «мусором»
int emptyArray[20];
// С инициализацией
// Массив из 10 вещественных чисел,
// первые два из них - 3.14 и 2.72, остальные нули
double partFilled[10] = { 3.14, 2.72 };
// Без размерности –
// будет создан массив из 4 элементов
bool filledArray[] = { true, true, false, true };
// Определение размера созданного массива
cout<<sizeof(filledArray)/sizeof(bool)<<endl;
// Инициализация массивов в стеке (автоматических)
// может работать по-разному в разных реализациях
```

Размерность массива – константа или переменная?

```
int arr[20];        // так можно
const int SIZE=20;
int arr1[SIZE];     // и так можно
int arr2[SIZE+5]; // и даже так
int length=func(SIZE);
int arr3[length]; // а так – НЕТ!
// точнее говоря, можно только в C99
// также это допускают некоторые
// компиляторы C++, хотя
// стандарт это не позволяет
```

Операция индексации []

```
// Пример заполнения целочисленного
// массива числами от 1 до 10
// …
int array[10];
for (int i=0; i<10; i++)
   // array[0]=1, array[1]=2, …
   array[i] = i+1;
// Индекс массива в C/C++ может
// изменяться от 0 до N-1, где
// N – размерность массива.
// При использовании неверного индекса,
// в лучшем случае произойдет
// ошибка при исполнении
```

Операция индексации []

```
// Пример подсчета суммы элементов
// целочисленного массива
int array[10];
// Заполнение массива
// …
int sum=0;
for (int i=0; i<10; i++)
    sum += array[i];
```

Пример: расчет первой 1000 простых чисел

- Как уже упоминалось ранее, для проверки числа X на простоту достаточно его делить не на все числа, а только на простые, причем достаточно проверить простые числа, меньшие или равные $X^{1/2}$
- Уже найденные простые числа нужно где-то хранить. Будем хранить их в целочисленном массиве primeArray

Пример: расчет первой 1000 простых чисел

```
#include <iostream>
#include <fstream>
using namespace std;
int main(void)
{
    ofstream out("prime.txt"); // Файл для вывода
    const int PRIME_NUM = 1000; // Всего нужно найти
    int primeFound = 0; // Уже найдено
    int primeArray[PRIME_NUM]; // Массив простых чисел
    int current = 2; // Текущее проверяемое число
    int limit = 1; // Последний проверяемый делитель
    bool isPrime = true; // Признак простоты
```

Пример: расчет первой 1000 простых чисел

```
while (primeFound < PRIME_NUM)
// Пока не нашли все числа
{
   if ((limit+1)*(limit+1)<=current)
      limit++;
   for (int i=0; i<primeFound &&
                     primeArray[i]<=limit; i++)
      if (current % primeArray[i]==0)
      // Если разделилось, то не простое
      {
         isPrime = false;
         break;
      }
   // ...
```

Пример: расчет первой 1000 простых чисел

```
    while (primeFound < PRIME_NUM)
    // Пока не нашли все числа
    {
        // ...
        if (isPrime)
        {
            out<<current<<endl; // Вывод
            // Сохранить, увелич. индекс
            primeArray[primeFound++]=current;
        }
        current++;
    }
    return 0;
}
```

Какие функции можно применить?

- Демонстрация в среде программирования

Обратите внимание!

- Если задать PRIME_NUM > 2^{18} (262144), то при исполнении программы произойдет ошибка «Stack overflow»
- Этой ошибки можно избежать, если разместить массив в статической памяти

Разновидность массива – строки языка С

- Строка с точки зрения языка С – это массив символов, заканчивающийся специальным нуль-символом (в языке С++ это также работает, но есть и другие варианты)
- Напоминание
  - тип **char**
  - коды символов

Определение строки/символьного массива

```
// Символьный массив из 10 ячеек
// Можно сохранить строку из 9 символов
char arr[10];
// Символьный массив со строкой hello
// Размерность массива - 6 элементов
// Используются двойные кавычки,
// причем нуль-символ не указывается
char str[] = "hello";
// Эквивалентно
char str[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
// Можно использовать escape-последовательности
char escstr[] = "hello, world!\n";
```

Ввод и вывод строк

- Выводить строки можно обычным способом, с помощью операции вывода <<
- Вводить строки также можно обычным образом. Однако, следует иметь в виду следующее:
  - пробельные символы (whitespaces) считаются при вводе концом строки; манипулятор noskipws на это не влияет
  - размер вводимой строки никак не контролируется; если пользователь введет строку большего размера, чем размерность массива, произойдет ошибка при исполнении (см. пример **IOString**) – а ведь пользователь всегда прав!

Ввод и вывод строк

- Поэтому, желательно вводить строки другими способами. Например:

```
char str[20];
// Читаем до 19 символов, или
// до перевода строки
cin.getline(str, 20);
// Читаем до 19 символов, или
// до символа ~
cin.getline(str, 20, '~');
// В сложных случаях применяется
// посимвольный ввод -
// вводятся отдельные символы строки
```

Операции со строками

- Стандартные операции **не могут** использовать строки в качестве аргументов. Вместо них **используются строковые функции**.

```
char str1[] = "Hello!";
char str2[20];
str2 = "World!"; // Нельзя!
str2 = str1; // И так нельзя!
str1 += "World!"; // И так тоже нельзя
```

Строковые функции

```
#include <string.h>
// Длина строки
int len = strlen(string1);
// Копирование string2 в string1
// Размер массива string1 не проверяется
strcpy(string1, string2);
// Сравнение строк на равенство
// Результат strcmp =0, если строки равны
if (!strcmp(string1, string2)) { … }
// Конкатенация (приписывание string2 к string1)
// Размер массива string1 не проверяется
strcat(string1, string2);
// И некоторые другие…
// См. string manipulation в MSDN
```

Безопасные строковые функции

```
#include <string.h>
char string1[20], string2[30];
// ...
// Копирование string2 в string1
// Не более 20 символов
strncpy(string1, string2, 20);
// Сравнение строк на равенство
// Не более 20 символов
if (!strncmp(string1, string2, 20)) { … }
// Конкатенация (приписывание string2 к string1)
// Приписывает 9 символов и нуль-символ
strncat(string1, string2, 9);
// И некоторые другие…
// См. string manipulation в MSDN
```

Прототипы строковых функций

```
char* strcpy(char s1[], char s2[]);
int strcmp(char s1[], char s2[]);
char* strcat(char s1[], char s2[]);
```

Строки C++ (краткий обзор)

```
#include <string>
using namespace std;
int main(void)
{
    string s1, s2;
    cin>>s1>>s2;
    s1 += s2;
    cout<<s2;
    return 0;
}
```

Пример работы со строкой

- Задано основание системы счисления (2<=p<=36) и записанное в ней целое число. В качестве цифр 10-35 используются латинские символы a-z или A-Z.
- Необходимо перевести число в десятичную систему счисления

Ход решения

- В общем случае мы не можем сохранить введенное число в виде целочисленной переменной, поэтому сохраним его в виде строки
- При анализе необходимо перебрать символы числа, переводя их в цифровое представление и умножая на p:
  $X=(((X_n*p+X_{n-1})*p+X_{n-2})*p+\ldots)*p+X_0$

Текст программы

```
#include <iostream>
#include <locale.h>
using namespace std;
int main(void)
{
   setlocale(LC_ALL, "Russian");
   cout<<"Перевод целых чисел из p-ичной с-мы в 10-ную"<<endl;
   int p;
   do
   {
      cout<<"Введите основание системы p (2...36): ";
      cin>>p;
   } while (p<2 || p>36);
   const int MAX_LEN = 30; // Максимальная длина числа
   cout<<"Введите число (не более "<<MAX_LEN<<" знаков): ";
```

Текст программы

```
    char x[MAX_LEN+1];
    // Первый раз для пропуска уже введенного перевода строки
    cin.getline(x, 1, '\n');
    // Второй раз собственно ввод
    cin.getline(x, MAX_LEN+1, '\n');
    int n=0; // Число в 10-ной системе
    for (int i=0; x[i]!=0; i++)
    // Перебор знаков числа до нуль-символа
    {
       // ...
    }
    cout<<"Число в десятичной системе: "<<n<<endl;
    return 0;
}
```

Текст программы

```
// Содержимое цикла for
n *= p;
int d=-1; // Цифра
if (x[i]>='0' && x[i]<='9')
   d=x[i]-'0'; // '0'...'9' => 0...9
else if (x[i]>='a' && x[i]<='z')
   d=10+x[i]-'a'; // 'a'...'z' => 10...35
else if (x[i]>='A' && x[i]<='Z')
   d=10+x[i]-'A'; // 'A'...'Z' => 10...35
if (d<0 || d>=p) // Проверка корректности
{
   cout<<"Цифра "<<x[i]<<" некорректна"<<endl;
   return 0;
}
n += d;
```

Проблемы

- Как бороться с переполнением?
  - Можно использовать константу INT_MAX - максимальное целое число; перед умножением и сложением можно контролировать возможность переполнения
- Ошибкой было бы задать цикл **for** так: (почему?)
  - **for** (**int** i=0; i<strlen(x); i++) { … }

Контроль переполнения -- подробнее

```
k=m+n;
if (m+n <= INT_MAX)
    k=m+n;
```

Контроль переполнения -- подробнее

```
k=m+n;
if (n <= INT_MAX-m) // Bingo!
    k=m+n;
// С умножением примерно аналогично
```

Какие функции можно применить?

- Демонстрация в среде программирования

Адреса

- Каждый байт памяти имеет свой целочисленный номер (**адрес**). Как правило, адреса записываются в 16-ной системе счисления
- У каждой переменной также есть свой адрес, для процессоров x86 он соответствует адресу **младшего байта** переменной (little-endian)

Указатели

- Указатель – особый тип переменной. В указателе хранится адрес некоторой другой переменной (говорят, что данная переменная **указывает** на другую).
- Тип указателя обозначается type*, где type – тип переменной, адрес которой хранит указатель. Например: **int***, **char***.
- Переменная, адрес которой мы храним, тоже может быть указателем – при этом у нас получается указатель на указатель, или двойной указатель. Его тип записывается как type**, например, **int****, **char****
- Аналогично возможны тройные, четверные и т.д. указатели

Адресация

- Для получения адреса переменной (а также, элемента массива, элемента структуры и пр.) используется операция **адресации** &x. Элемент выражения, имеющий адрес, называется **L-value** (см. также лекцию 2).

```
int n=5;
int* ptr=&n;
```

- Регистровые переменные адреса не имеют.

```
register int n=5;
int* ptr=&n; // Подобная операция запрещена
```

- Обратите внимание: чтобы получить из типа указатель на тип, используется *, а чтобы получить из переменной адрес переменной, используется &

Разадресация

- Для получения по адресу переменной ее значения используется операция **разадресации** *ptr, обратная операции адресации: *(&x) всегда равно x.

```
int n=5;
int* ptr=&n;
cout<<*ptr; // Выведется 5
```

Следует иметь в виду, что…

- Если указатель не инициализирован (его начальное значение не указано), его разадресация приведет к ошибке (так как в нем хранится случайный адрес)

```
int* ptr;
cout<<*ptr; // Ошибка!
```

- Указатель, хранящий нулевой адрес, не указывает ни на что. Его разадресация приводит к ошибке, но эта возможность часто используется, например, для обозначения несуществующих объектов

```
int* ptr=0; // Вместо 0 может использоваться NULL
cout<<*ptr; // Ошибка!
```

Связь массивов с указателями

- Имя массива может быть использовано как указатель на его элемент с индексом 0:

```
char str[] = "Hello!";
char* ptr=str; // ptr указывает на H
cout<<*ptr<<*str; // Выведется HH
```

- При увеличении указателя на 1 он передвигается на следующий элемент массива, при уменьшении - на предыдущий:

```
ptr++; // Теперь ptr указывает на e
cout<<*ptr<<*(ptr-1); // Выведется eH
```

- Аналогично, можно увеличить или уменьшить указатель на любое целое число, передвигая его на соответствующий элемент массива:

```
cout<<*(str+5); // Выведется !
```

Связь указателей с массивами

- С другой стороны, любой указатель может быть использован как имя начала массива соответствующего типа. К нему применима операция индексации:

```
char str[] = "Hello!";
char* ptr=str+2; // Указываем на 1-ю l
cout<<ptr[0]<<ptr[2]; // Выведется lo
```

- Указатель на символ может быть использован как имя начала строки:

```
cout<<ptr; // Выведется llo!
```

- Вообще, ptr[i] эквивалентно *(ptr+i), независимо от того, является ptr указателем или именем массива
- Аргументы функций: type* то же, что и type[]

Пример: копирование строки разными способами

```
#include <iostream>
using namespace std;
int main(void)
{
   char src[] = "Some string";
   char dst[20];
   // Копирование через индексацию
   int i;
   for (i=0; src[i]!=0; i++)
      dst[i]=src[i];
   dst[i]=0;
   cout<<dst<<endl;
```

Пример: копирование строки разными способами

```
    // Копирование через указатели I
    const char* psrc=src; // Константный указатель
    char* pdst=dst;
    while (*psrc)
        *pdst++ = *psrc++;
    *pdst = 0;
    cout<<dst<<endl;
    // Копирование через указатели II
    psrc=src;
    pdst=dst;
    while (*pdst++ = *psrc++); // Пустой цикл!
    cout<<dst<<endl;
    return 0;
}
```

Одномерные и многомерные массивы

- Рассмотренные нами массивы являлись одномерными или линейными – у них одна размерность
- Язык С допускает двумерные, трехмерные и бОльшие массивы:

```
int array2d[3][4]; // Двумерный
int array4d[5][8][12][4]; // Четырехмерный
```

- При инициализации, используется несколько уровней вложенных { }. Не указывать можно только первую размерность:

```
int array2d[][2] = { { 1, 2 } , { 3, 4}, {5, 6} };
```

Двумерные массивы в памяти

- Двумерные массивы arr[M][N] хранятся в памяти в следующем порядке:
  - [0][0], [0][1], …, [0][N-1], [1][0], [1][1], …, [1][N-1], …, [M-1][0], [M-1][1], …, [M-1][N-1]
  - то есть, сначала хранится 0-я строка, потом 1-я, 2-я и так далее.
- Имя массива с индексом arr[i] может быть использовано как указатель на элемент [i][0], или как имя одномерного массива
- Имя массива без индекса arr в данном случае не может быть использовано как указатель

Пример: транспонирование матрицы

- В файле in.txt хранится матрица A (двумерная таблица) в формате:

  M N

  a_{11} a_{12} … a_{1N}

  a_{21} a_{22} … a_{2N}

  …

  a_{M1} a_{M2} … a_{MN}

- Где M – число строк, N – число столбцов, a_{ij} - вещественные элементы матрицы

Пример: транспонирование матрицы

- Необходимо вывести в файл out.txt транспонированную матрицу (сделать строки столбцами и наоборот):
  - a_{11} a_{21} … a_{M1}
  - a_{12} a_{22} … a_{M2}
  - …
  - a_{1N} a_{2N} … a_{MN}

Ход решения

- Для хранения матрицы используем двумерный массив
- По правилам C/C++, размер массива должен быть константой! Поэтому, мы вынуждены ограничить максимальный размер матрицы, например, числом 20. При этом мы храним массив [20][20], а реально используем только часть элементов

Текст программы

```
#include <iostream>
#include <fstream>
#include <locale.h>
using namespace std;
int main(void)
{
   setlocale(LC_ALL, "Russian");
   const int MAX_SIZE=20;
   int m, n;
   double matrix[MAX_SIZE][MAX_SIZE];
   ifstream in("in.txt");
   if (!in.is_open())
   {
      cout<<"Файл in.txt не существует"<<endl;
      return -1;
   }
```

Текст программы

```
in>>m>>n;
if (in.fail() || m<1 || m>MAX_SIZE ||
    n<1 || n>MAX_SIZE)
{
   cout<<"Некорректные размерности"<<endl;
   return -2;
}
for (int i=0; i<m; i++)
   for (int j=0; j<n; j++)
      in>>matrix[i][j];
if (in.fail())
{
   cout<<"Не удалось прочитать матрицу"<<endl;
   return -3;
}
```

Текст программы

```
    ofstream out("out.txt");
    if (!out.is_open())
    {
        cout<<"Не удалось создать файл out.txt"<<endl;
        return -4;
    }
    for (int j=0; j<n; j++)
    {
        for (int i=0; i<m; i++)
            out<<matrix[i][j]<<" ";
        out<<endl;
    }
    cout<<"Программа завершена успешно"<<endl;
    return 0;
}
```

Основная проблема

- Мы вынуждены всегда создавать массив максимально возможного размера
- В условии не сказано, что таблица не может быть больше, чем 20x20
- Хорошо бы все же задавать массив переменного размера…
  - Если у нас C99 – это легко (см. Слайд 4)
  - Иначе у нас некоторые проблемы

Решение в C/C++ – динамическая память

- Динамическая память выделяется по требованию пользователя, с помощью оператора **new**, и освобождается также по требованию пользователя, с помощью оператора **delete**
- Размер динамической памяти задается при ее выделении, например (для массива):

```
int size;
cin>>size; // ptr будет указывать на начало массива
int* ptr=new int[size];
…
delete[] ptr; // Освобождаем выделенный массив
```

- Или, для одного элемента:

```
int* ptr=new int;
…
delete ptr; // Освобождаем выделенный элемент
```

- В C вместо new/delete используются функции malloc/free (см. MSDN)

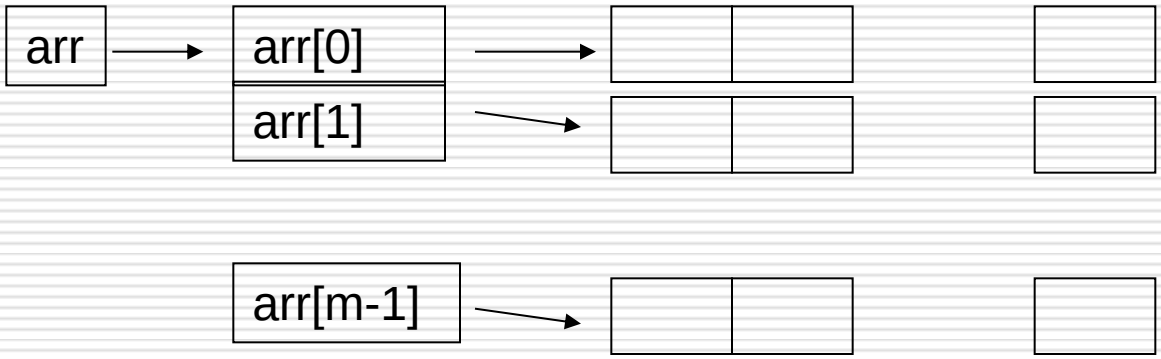
Как быть с двухмерным массивом?

- Может быть, так?

```
int* arr=new int[m][n]; // Так нельзя…
```

- Можно разместить в динамической памяти массив указателей на одномерные массивы:

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



Как быть с двумерным массивом?

- Второй вариант - это разместить в динамической памяти одномерный массив размером m*n. Однако, в этом случае нам придется самим рассчитывать его индексы. Элемент i-й строки и j-го столбца будет расположен по индексу i*n+j (здесь строки и столбцы нумеруются с 0).

Текст программы (модифицированный)

```
#include <iostream>
#include <fstream>
#include <locale.h>
using namespace std;
int main(void)
{
   setlocale(LC_ALL, "Russian");
   int m, n;
   ifstream in("in.txt");
   if (!in.is_open())
   {
      cout<<"Файл in.txt не существует"<<endl;
      return -1;
   }
```

Текст программы

```
in>>m>>n;
if (in.fail() || m<1 || n<1)
{
   cout<<"Некорректные размерности"<<endl;
   return -2;
}
double** matrix = new double*[m];
for (int i=0; i<m; i++)
   matrix[i] = new double[n];
for (int i=0; i<m; i++)
   for (int j=0; j<n; j++)
      in>>matrix[i][j];
if (in.fail())
{
   cout<<"Не удалось прочитать матрицу"<<endl;
   return -3;
}
```

Текст программы

```
ofstream out("out.txt");
if (!out.is_open())
{
   cout<<"Не удалось создать файл out.txt"<<endl;
   return -4;
}
for (int j=0; j<n; j++)
{
   for (int i=0; i<m; i++)
      out<<matrix[i][j]<<" ";
   out<<endl;
}
```

Текст программы

```
    // Освобождаем в обратном порядке
    for (int i=0; i<m; i++)
       delete[] matrix[i];
    delete[] matrix;
    cout<<"Программа завершена успешно"<<endl;
    return 0;
}
```

Оставшиеся вопросы

- А что, если памяти не хватит?
  - Выполнение оператора **new** может вызвать ошибку при исполнении при недостатке памяти. Ее можно обработать – но способ обработки мы рассмотрим позже

Оставшиеся вопросы

- Хорошо бы не задавать во входном файле размеры матрицы…
  - Так можно сделать, но тогда нам придется посчитать числа в файле самим и создать в памяти матрицу нужного размера. Желающие могут найти решение самостоятельно

Оставшиеся вопросы

- Передача матриц в функции
- Для ступенчатого массива обычно передаётся двойной указатель И размеры

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

Оставшиеся вопросы

- Функция, создающая матрицу
- Проще всего матрицу вернуть как результат (что забыли?)

```
double** readMatrix(char* fname) {
    ifstream in(fname);
    if (!in.is_open()) return 0; // Failed
    int rows, cols;
    in >> rows >> cols;
    double** matrix = new double*[rows];
    for (int i=0; i<rows; i++) {
        matrix[i] = new double[cols];
        for (int j=0; j<cols; j++) in >> matrix[i][j];
    }
    return matrix;
}
```

Версия с размерностями

```
double** readMatrix(char* fname, int& rows, int& cols) {
    ifstream in(fname);
    if (!in.is_open()) return 0;
    in >> rows >> cols;
    double** matrix = new double*[rows];
    for (int i=0; i<rows; i++) {
        matrix[i] = new double[cols];
        for (int j=0; j<cols; j++) in >> matrix[i][j];
    }
    return matrix;
}
```

Чем ссылка отличается от указателя?

- Только C++ (в C ссылок нет)
- При определении ссылки, в имени типа используется символ & вместо *
- Для получения ссылки, не требуется явно проводить адресацию

```
int i, j;      // сами переменные
int* ptr=&i; // указатель, явная адресация
int& ref=i;  // ссылка, неявная адресация
```

- Для получения значения, на которое ссылаемся, не требуется явно проводить разадресацию

```
*ptr=0; // явная разадресация
 ref=0; // неявная разадресация
```

Чем ссылка отличается от указателя?

- Указатель можно перенаправить на другую переменную, а ссылку нет

```
ptr=&j; // теперь ptr указывает на j
ref=j;  // ref все равно ссылается на i
// и значение i теперь совпадает с j
```

- Указатель может не указывать никуда (быть равным 0 или просто «висеть в воздухе»). Ссылка всегда на что-то ссылается

```
int* ptr2; // висячий указатель
ptr2=0; // нулевой указатель
int& ref2; // так нельзя - нет инициализации
int& ref2=j;
ref2=0; // 0 будет записан в j
```

Аргументы главной функции

- Главная функция может обрабатывать так называемые аргументы командной строки, при этом программа запускается в следующем формате:
  - program.exe arg1 arg2 arg3
- Из среды программирования аргументы командной строки тоже можно задать – в настройках проекта (Command Line Arguments)

Аргументы главной функции

- **int** main(**int** argc, **char**** argv) { … }
  - argc – число аргументов, включая имя исполняемого файла
  - argv – указатель на массив указателей на строки с аргументами (используется аналог ступенчатого массива)
    - argv[0] – всегда имя исполняемого файла
    - argv[1] – 1-й аргумент
    - argv[2] – 2-й аргумент
    - …
    - argv[argc-1] – последний аргумент

Для чего можно использовать аргументы командной строки?

- Например, можно в них указать входной и выходной файл программы
- Рассмотрим программу, которая читает имена этих двух файлов из командной строки

Текст программы

```
#include <iostream>
#include <locale.h>
using namespace std;
int main(int argc, char** argv)
{
   setlocale(LC_ALL, "Russian");
   if (argc < 3)
   {
      cout<<"Строка запуска: "<<
         " MainArgs.exe infile outfile"<<endl;
      return -1;
   }
   cout<<"Входной файл: "<<argv[1]<<endl;
   cout<<"Выходной файл: "<<argv[2]<<endl;
   return 0;
}
```

Итоги

- Рассмотрены
  - Массивы
  - Массивы символов = строки
  - Указатели и их связь с массивами
  - Адресация / Разадресация
  - Многомерные массивы
  - Массивы в динамической памяти
  - Ссылки
  - Аргументы командной строки
- Далее
  - Некоторые функции библиотеки С