# НРТК

# C / C++

## Тема 08. Указатели и массивы. Ссылки и функции

# Указатели

# Указатель на <тип>

- Переменная типа «указатель на **<тип>**» — переменная, способная хранить адреса переменных с типом, указанным при ее объявлении
- Синтаксис:

```
<тип>* <идентификатор>;
```

- Пример:

```
int* pi;      // pi – указатель на int
double* pd;      // pd – указатель на double
char* pc;     // pc – указатель на char
```

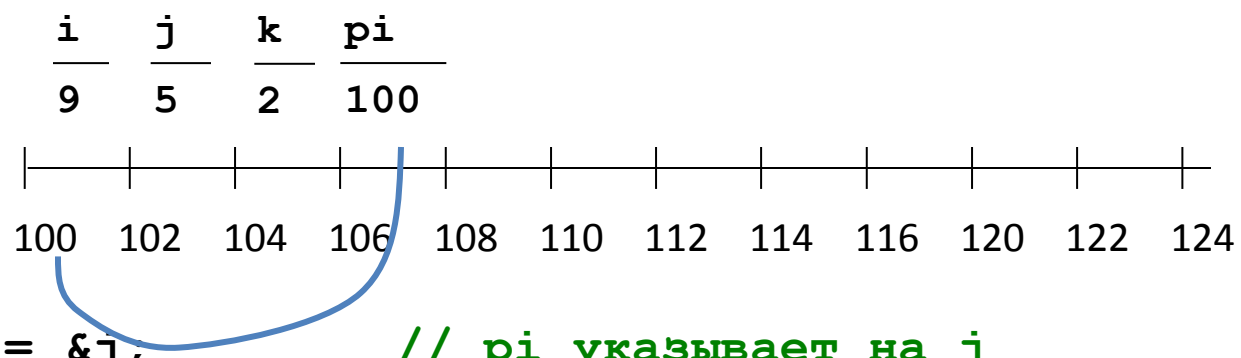
- Будьте внимательны:

```
int* a, b;       // a – int*, b – int
int* a, *b;      // a – int*, b – int*
```

# Значение «указатель на <тип>»

- Значение переменной типа «указатель на `<тип>`» — адрес объекта типа `<тип>`

```
int i = 9, j = 5, k = 2;
int* pi = &i;        // pi указывает на i
```




```
pi = &j;             // pi указывает на j
```

- `&` — операция взятия адреса
- Нулевой указатель — «никуда» не указывает

```
pi = 0;
```

# Операции с указателями

# Операция разыменовывания

- Разыменовывание — обращение к переменной, на которую указывает указатель
- Синтаксис:

  ***<указатель>**
- Тип у этого выражения — **<тип>** из объявления указателя
- Пример:

```
int i = 9, j = 5, k = 2;
int* pi = &i;
*pi = 3;          // i = 3
pi = &j;
k = i + *pi;      // k = 8
```

# Сложение, вычитание

- Сложение, вычитание с целым

```
<тип> var, *p = &var;
p = p ± i;      // p = &var ± i*sizeof(<тип>)
```

- Пример:

```
int ms[20], *p;
p = &ms[0];
*(p + 3) = 8;  // *(p + 3) ≡ ms[3]
```

- Вычитание указателей

```
<тип> vr1, vr2 , *p1 = &vr1, *p2 = &vr2;
i = p1 – p2; // i = (&vr1 - &vr2)/sizeof(<тип>)
```

- Пример:

```
int ms[20], *p1 = &ms[3], *p2 = &ms[8], i;
i = p1 – p2;   // i = -5
```

# Инкремент, декремент

- Инкремент

```
<тип> var, *p = &var;
p = p++;     // p = p + 1
```

- Декремент

```
<тип> var, *p = &var;
p = p--;     // p = p - 1
```

# Сравнение

- Указатели можно сравнивать друг с другом, при этом сравниваются адреса:

```
int i1, i2;
int* pi1 = &i1, *pi2 = &i2;
if(pi1 < pi2) ...    // if(pi1 - pi2 < 0)
```

- Указывает ли указатель на что-нибудь:

```
if(p != 0)...
if(p)
```

# Указатель void*

- Указатель на объект любого типа можно присвоить переменной типа `void*`, указатели типа `void*` можно сравнивать друг с другом
- Указатель `void*` — указатель на «сырую память», не имеющую типа, его нельзя разыменовывать, использовать в арифметических операциях и др.
- Пример:

```
int i, *pi; double* pd;
void* p = &i;              //Верно
pi = static_cast<int*>(p);    //Верно
pd = static_cast<double*>(p);//Верно, но...
```

# Указатели и константы

- Указатель на константу не может менять значение объекта, на который указывает, но может быть переуказан на другой объект
- Синтаксис:

```
const <тип>* <идентификатор>
```

- Пример:

```
int i = 9, j = 5, k = 4;
const int* pi;
pi = &i;
*pi = 3;     // Ошибка компиляции
j = *pi + k;// Допустимо
pi = &j;     // Допустимо
```

# Указатели и константы

- Константный указатель может менять значение объекта, на который указывает, но не может быть переуказан на другой объект
- Синтаксис:

```
<тип>* const <идентификатор>
```

- Пример:

```
char c = 5;
char* const win_core = &c;
char ch;
win_core = &ch;// Ошибка компиляции     *win_core = 8;  // c = 8
```

# Указатели и константы

- Константный указатель на константу не может менять значение объекта, на который указывает, не может быть переуказан на другой объект
- Синтаксис

```
const <тип>* const <идентификатор>
```

Пример

```
const double pi = 3.1415926535;
const double* const pointer_to_pi = &pi;
```

# Запрос и освобождение памяти

# C++: операции new, delete

- Запрос памяти во время выполнения программы
  - один экземпляр

```
<тип>* <указатель> = new <тип>;
```

  - *n* экземпляров

```
<тип>* <указатель> = new <тип>[n];
```

- Освобождение памяти
  - один экземпляр

```
delete <указатель>;
```

  - *n* экземпляров

```
delete[] <указатель>;
```

# Пример

```
int* ms = new int[20];

for (int i = 0; i < 20; i++)
  ms[i] = i;

delete[] ms;

ms = new int[10];
```

# C: функции malloc и free

- Запрос *n* экземпляров переменных во время выполнения программы

```
#include <stdlib.h>
...
<тип>* <указатель> = [(<тип *>)]
malloc(n*sizeof(<тип>));
```

- Освобождение памяти

```
free(<указатель>);
```

# Пример

```
int i;
int* ms;

ms = (int*)malloc(20 * sizeof(int))

for (i = 0; i < 20; i++)
  ms[i] = i;

free(ms);

ms = (int*)malloc(10 * sizeof(int));
```

# Резюме

- Память, выделенная таким образом называется *динамически выделенной*
- Работа с указателями и динамически выделяемой памятью вообще — один из самых сложных моментов в языках C и C++
- Указатель способен хранить адрес не только одной ячейки памяти (переменной), но и адрес блока некоторых смежных ячеек памяти (массива)
- Временем жизни выделенной динамически памяти управляет программист — выделение памяти и ее возврат в систему выполняются явно

# Типичные ошибки

- Повисшие ссылки

```
int* p = new int[256];
...
int* q = p;
...
delete[] p;     // Что с q?
```

- Утечки памяти

```
{
  int* p = new int[256];
  ...
}          // Как теперь удалить массив?
```

# Пример: работа с массивами

```
int* array;
int sum, n, i;

do
{
  cout << "Введите количество элементов массива:";
  cin >> n;
} while ( n <= 0 );

array = new int[n];
for ( i = 0; i < n; i++ )
{
  cout << "[" << i << "] = ";
  cin >> array[i];
}
delete[] array;
```

# Массивы и указатели

# Массив = указатель

- Идентификатор массива — фактически константный указатель на массив
- Идентификатор массива – параметра функции — неконстантный указатель на массив
- Примеры:

```
int i;
int ma[20];
for ( i = 0; i < 20; i++ )
  ma[i] = i;// *(ma + i) = i;  i[ma] = i;

for ( i = 0; i < 20; i++ )
     *ma++ = i;// Ошибка: Lvalue required

char s1[] = "Строка1";
char s2[] = "Строка2";
if(s1 == s2)...// Сравнение адресов, не строк!
```

# Пример: StrLen, StrCopy

```
size_t StrLen(const char* str)
{
  const char* p = str;
  while ( *p++ );
  return p - str - 1;
}

char* StrCopy(const char* source)
{
  char* tmp = new char[StrLen(source) + 1];
  while ( tmp++ = source++ ) ;
  return tmp;
}
```

# Выделение памяти в теле функции

- Кто ответственен за освобождение выделенной памяти? — тот, кто выделил память, тот и должен ее удалять!
- Решения
  - аккуратный выбор идентификаторов

```
char* CreateStr(const char* source);
```

  - возврат выделенной памяти только через параметры (рассматривается позже в этой теме):

```
void CloneStr(char*& dest, const char* source);
```

  - функции не возвращают указателей на выделенную в них память

```
char* StrCopy(char* dest, const char* source);
```

# Пример: замена символа в строке

```
int ReplSym(const char* str, char sym1, char sym2)
{
  const char* p = str;
  int cnt = 0;

  while ( *p )
  {
    if ( *p == sym1 )
    {
      *p = sym2;
      cnt++;
    }
    p++;
  }
  return cnt;
}
```

# Пример: StrToInt

```
bool StrToInt(int& num, const char* str)
{
  num = 0;
  while(*str)
  {
    if(*str < '0' || *str > '9')
      return false;

    num = num * 10 + *str++ - '0';
  }

  return true;
}
```

# Массивы указателей, указатели на массивы

- Синтаксис объявления массива из указателей:

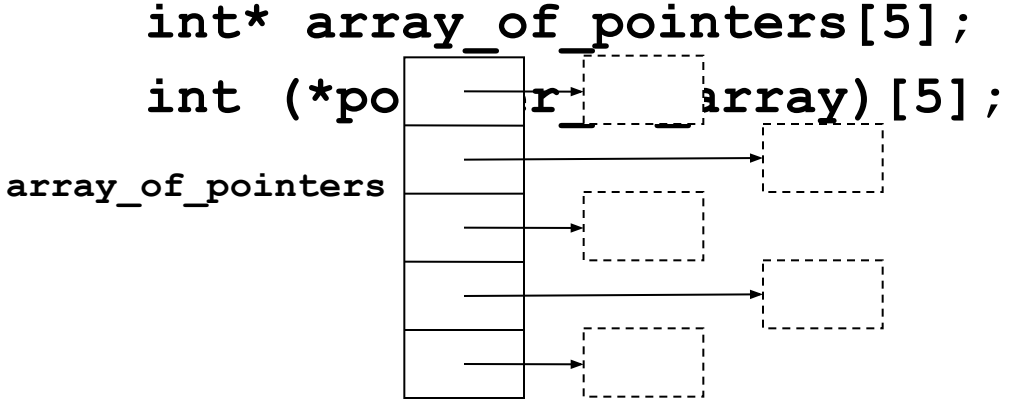
```
<тип>* <имя>[<размер>];
(<тип>*) <имя>[<размер>];
```

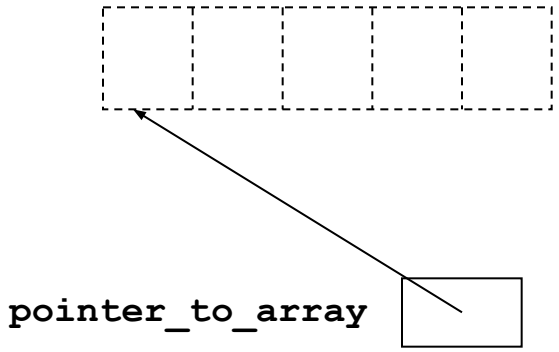
- Синтаксис объявления указателя на массив:

```
<тип> (*<имя>)[<размер>];
```

- Примеры:

```
int* array_of_pointers[5];
int (*po    r_   array)[5];
```



# Двумерный массив

- Идентификатор двумерного массива — указатель на массив указателей на строки (столбцы) массива

```
int a[5][4];
for ( i = 0; i < 5; i++ )
  for ( j = 0; j < 4; j++ )
    a[i][j] = 10 * i+j;
    *(a[i] + j) = 10 * i+j;        // или так
    *(*(a + i) + j) = 10 * i+j;    // или так
```

- Функция, принимающая на вход двумерный массив, может быть только примерно такой:

```
void print_matrix(double m[][20], int n);
```

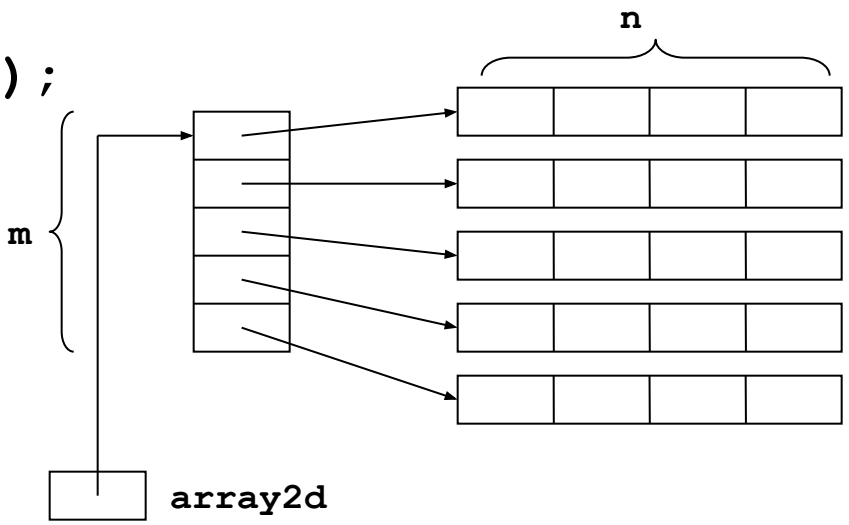
Или такой:

```
void print_matrix1(double **m, int n, int m);
```

# Двумерный массив

- Динамическое построение двумерного массива

```
int m = 5, int n = 4;
int** array2d;
array2d = new int*[m];
for ( i = 0; i < m; i++ )
  array2d[i] = new int[n];

print_matrix1(array2d, m, n);

for ( i = 0; i < m; i++ )
  delete[] array2d[i];
delete[] array2d;
```



# Двумерный массив

- Функция, использующая динамический двумерный массив:

```
void print_matrix1(double** a, int m, int n)
{
  for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)
      cout << a[i][j];
}
```

но лучше так:

```
void print_matrix1(const double** a, int m, int n);
```

а еще лучше так:

```
void print_matrix1(const double* const* a,
          int m, int n);
```

# Трехмерный массив

- Идентификатор трехмерного массива — указатель на массив указателей на массивы указателей на строки (столбцы) массива

```
int n1 = 3, n2 = 4, n3 = 5;
int*** array3d;
array3d = new int**[n1];
for ( int i = 0; i < n1; i++ )
{
  array3d[i] = new int*[n2];
  for ( int j = 0; j < n2; j++ )
    array3d[i][j] = new int[n3];
}
//...
```

# Трехмерный массив

```
//...
for ( i = 0; i < n1; i++ )
{
  for ( int j = 0; j < n2; j++ )
    delete[] array3d[i][j];
  delete[] array3d[i];
}
delete[] array3d;
```

# Пример: ввод текста

```
const maxl = 80;
char** txt, buf[maxl + 1], *pb, *ps, **pt;
int nt;
cout << "К-во строк текста = ";
cin >> nt;
txt = new char*[nt];
pt  = txt;
for ( int i = 0; i < nt; i++ )
{
  cin.getline(buf, 80);
  *pt = new char[StrLen(buf)]; // pt[i] = new …
  pb = buf;
  ps = *pt++;          // ps = pt[i];
  while ( *ps++ = *pb++ ) ;
}
```

# Задачи

- Вывод на консоль текстов, содержащих кириллицу
- Функции работы со строками

# Ссылки и функции

# Описание функции

- Синтаксис:

```
<тип> <имя_функции>(<параметры>)
{
  <тело функции>
  return <возвращаемое значение>;
}
```

- где:
  - `<параметры>` — список исходных данных (формальных параметров) в виде:
    `<тип> имя_переменной`
  - `<тело функции>` — описание алгоритма функции
  - `<возвращаемое значение>` — результат, возвращаемый из функции, типа `<тип>`

# Вызов функции

- Синтаксис:

  **<имя_функции> (<передаваемые параметры>)**

- где:
  - **<передаваемые параметры>** — список передаваемых исходных данных (фактических параметров); элементом списка может быть:
    - константа
    - переменная
    - выражение
- В C++ количество и типы фактических параметров должны совпадать с количеством и типами формальных параметров

# Функция, не возвращающая значения

- Если функция не должна возвращать никакого значения, то используется тип `void`, `return` —необязателен
- Пример: вывод треугольника из звездочек:

```
void print_triangle(int n)
{
  for (int i = 0; i < n; i++ )
  {
    for (int j = 0; j < i + 1; j++ )
      cout << "* ";
    cout << endl;
  }
}
```

Демонстрация

# Функция без параметров

- Если функция не получает параметров, то список аргументов — пустой (скобки остаются)
- Пример (вычисление машинного $\varepsilon$ для `double`):

```
double double_eps()
{
  double epsilon = 1.0;
  while(epsilon / 2.0 + 1 > 1)
    epsilon /= 2.0;
  return epsilon;
}
```

- Вызов функции:

```
cout << double_eps() << endl;
```

# Механизм получения параметров

- Каждый формальный параметр является локальной переменной функции
- При вызове функции значения фактических параметров заносятся в соответствующие им формальные параметры — *передача параметра по значению*
- После выполнения функции значения (возможно измененных) формальных параметров назад не возвращаются
- Что делать если измененные значения *нужно* вернуть из функции?

# Возврат значений через параметры в C

# Передача параметров по указателю

- Для возврата значения параметр должен быть объявлен как указатель
- Фактический параметр не может быть выражением или константой, только переменной (тем, у чего можно взять адрес и что можно изменять)
- Такой механизм передачи параметров называется *передачей параметра «по указателю»* (хотя, фактически, это тоже возврат «по значению»)

# Как вернуть переменную?

```
// Получаем указатель на переменную
void Sum(int a, int b, int* s)
{
  // Присваиваем разыменованному указателю
  *s = a + b;
}

int main()
{
  int a;
  Sum(2, 3, &a); // Передаем адрес переменной
  return 0;
}
```

# Как вернуть указатель?

```
// Получаем указатель на указатель
void Input(char** ss)
{
  char* tmp = (char*)malloc(20);
  gets(tmp);
  // Присваиваем разыменованному указателю
  *ss = tmp;
}

int main()
{
  char* str;
  Input(&str); // Передаем адрес указателя
}
```

# Возврат значений через параметры в C++. Ссылки и функции

# C++: ссылка на переменную

- В C++ помимо указателей есть еще один тип — «ссылка на переменную»
- Синтаксис:

```
<тип>& <идентификатор> = <инициализатор>
```

  где:

  - `<идентификатор>` — идентификатор ссылки
  - `<инициализатор>` — идентификатор переменной, на которую ссылка ссылается
- Ссылка на переменную — новая переменная, являющаяся псевдонимом переменной, на которую «ссылается ссылка»

# C++: ссылка на переменную

- Ссылка должна быть обязательно инициирована
- Примеры:

```
int i = 2, j = 3;
int& ri = i;    // ri – ссылка на i
ri = 8;         // i = 8
int& rr;        // ошибка – нет инициализации
ri = &j;        // ошибка – это не указатель
```

# Передача параметров по ссылке

- Для возврата значения параметр должен быть объявлен как ссылка
- Фактический параметр не может быть выражением или константой, только переменной (тем, у чего можно взять адрес и что можно изменять)
- Такой механизм передачи параметров называется *передачей параметра «по ссылке»*

# Как вернуть переменную в C++?

```
// Получаем ссылку на переменную
void Sum(int a, int b, int& s)
{
  // Присваиваем ссылке
  s = a + b;
}

int main()
{
  int a;
  Sum(2, 3, a); // имя переменной – это ссылка на нее
}
```

# Как вернуть указатель в C++?

```
// Получаем ссылку на указатель
void Input(char*& ss)
{
  const maxl = 20 + 1;
  char* tmp = new char[maxl];
  cin.getline(tmp, maxl);
  // Присваиваем ссылке
  ss = tmp;
}

int main()
{
  char* str;
  Input(str); // Передаем ссылку на указатель
}
```

# Передача параметров по ссылке

- Передача параметров по ссылке используется
  - для возврата из функции нескольких значений
  - для того, чтобы не копировать громоздкий объект (прежде всего относится к структурам и классам — позже)
- Нет способа передать массив «по значению», т. е. копируя его

# Пример: сокращение дроби

```
int gcd(int a, int b);

void simplify(int& a, int& b)
{
  int p = gcd(a, b);
  a /= p;
  b /= p;
}
```

# Пример: функция ввода строки

```
int GetLine(char*& str)
{
  const int maxl = 80 + 1;
  char buf[maxl],
  cout << "Введите строку: ";
  cin.getline(buf, maxl);
  int nst = StrLen(buf);
  char *p = new char[nst + 1];
  str = p;
  while ( *p++ = *buf++ ) ;
  return nst;
}
```

# «По значению» vs. «по ссылке/указателю»

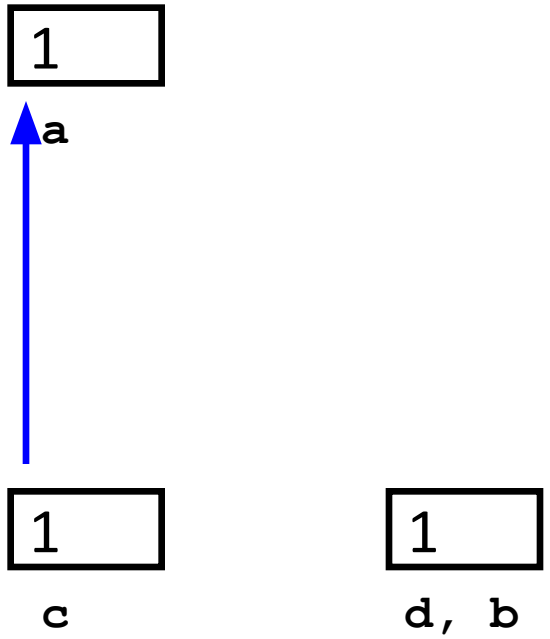
- Основная функция: `c = 1, d = 1`

```
void foo(int a, int& b)
{
  a = 2;
  b = 2;
}

int main()
{
  int c = 1, d = 1;
  foo(c, d);
  cout << c << d;
  return 0;
}
```

# «По значению» vs. «по ссылке/указателю»

- Вызов функции: копирование и ссылка

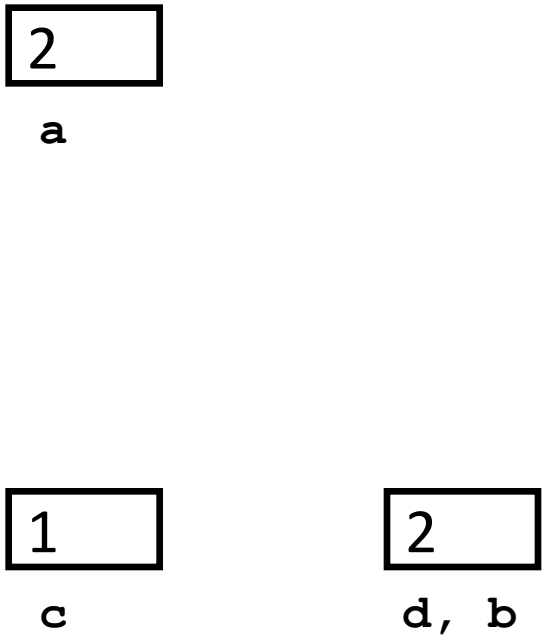
```
void foo(int a, int& b)
{
  a = 2;
  b = 2;
}

int main()
{
  int c = 1, d = 1;
  foo(c, d);
  cout << c << d;
  return 0;
}
```



# «По значению» vs. «по ссылке/указателю»

- Изменение значения: `c = 1, d = 2`

```
void foo(int a, int& b)
{
  a = 2;
  b = 2;
}

int main()
{
  int c = 1, d = 1;
  foo(c, d);
  cout << c << d;
  return 0;
}
```



# «По значению» vs. «по ссылке/указателю»

- Возврат из функции

```
void foo(int a, int& b)
{
  a = 2;
  b = 2;
}

int main()
{
  int c = 1, d = 1;
  foo(c, d);
  cout << c << d;
  return 0;
}
```

# Возможности функций в C++

# Перегрузка функций

- Перегрузка — использование нескольких функций с одним именем, но разными наборами формальных параметров (*сигнатурами*)
- Компилятор выбирает подходящую функцию по типам фактических параметров

# Пример: минимум из двух чисел

```
int min(int a, int b)           { return (a < b) ? a : b; }
double min(double a, double b) { return (a < b) ? a : b; }
int min(int a, int b, int c)
{
  return (a < b) ? min(a,c) : min(b,c);
}
int main()
{
  int a, b;
  double c;
  a = min(2, 7);
  b = min(5, 3, 7)
  c = min(9, 7.8);  // Какую версию выбрать?
  return 0;
}
```

# Правило выбора перегруженной функции

- Точное соответствие количества и типов параметров
- Соответствие с учетом встроенных преобразований типов для параметров функции:

  **`char, short → int → long`**

  **`float → double → long double`**
- Соответствие с учетом определенных программистом преобразований типов для параметров функции
  - это про классы (будет далее)
- Тип возвращаемого значения не учитывается

# Параметры по умолчанию

- Возможность задавать значения формальных параметров при описании функции
- При вызове таких функций параметры, имеющие значения по умолчанию можно не указывать
- Только самые правые параметры из списка могут быть параметрами по умолчанию

# Пример: замена символов в строке

```
void repl(char* st, char c1 = 'A', char c2 = 'a')
{
  while ( *st )
  {
    if (*st == c1) *st = c2;
    *st++;
  }
}
int main()
{
  char ss[50] = "АбрА кАдАбрА бум";
  repl(ss);
  repl(ss, 'p');
  repl(ss, 'a', 'O');
}
```

# Пример: расстояние между точками

```
double dist(double x1, double y1,
  double x2 = 0.0,double y2 = 0.0)
{
  return sqrt( (x1 - x2)*(x1 - x2) +
               (y1 - y2)*(y1 - y2) );
}


...
// Расстояние от (10, 10) до (20, 20)
dist(10.0, 10.0, 20.0, 20.0);
// Расстояние от (15, 15) до начала координат
dist(15.0, 15.0);
```

# Встраиваемые функции

- Объявление функции как `inline` (встраиваемой) — рекомендация компилятору встроить тело функции в место ее вызова:

```
inline <тип> имя(<параметры>) <тело функции>
```

- Пример:

```
        double d3 (double a)   { return a*a*a; }
inline double d3i(double a)   { return a*a*a; }

int main()
{
  cout << d3(2.1);
  cout << d3i(2.1);
  return 0;
}
```

# C/C++: макросы с параметрами

- Похожий результат можно получить с помощью:

```
#define <идентификатор>(<параметры>) <строка>
```

- Пример:

```
#define D3(X) X*X*X
int main()
{
  double a = 3.4, b;
  b = D3(a); // Преобразуется в  b = a*a*a
}
```

- Опасно:

```
b = D3(a + 1); // b = a + 1*a + 1*a + 1
```

надо было:

```
#define D3(X) ((X)*(X)*(X))
```

# C/C++: еще пример макроса

```
#define DETERM(A, B, C) ((B)*(B) - 4.*(A)*(C))

int main()
{
  double a, b, c, d;
  if ( ( d = DETERM(a, b, c) )  > 0. )
  ...
}
```

# C/C++: другие возможности

- Функции с переменным числом аргументов

```
int printf(const char* format, ...);
```

- Параметры без имени

```
double dist(double x1, double y1, double x2,
double y2, double)
{
   ...
}
```

# C и C++: имя функции как указатель на функцию

# Тип «указатель на функцию»

- Синтаксис:

```
<тип> (*имя)(<параметры>)
```

- Объявляет новый тип — указатель на функцию (способен хранить адрес функции)
- Пример:

```
double f(double x);

double (*g)(double) = &f;
// Обычно:
typedef double (*fun1d)(double);
fun1d g = &f;
...
f(2.0);
(*g)(2.0);
```

# Функция как указатель

- Имя функции неявно приводится типу указатель на функцию и обратно
- Благодаря этому можно:
  - передавать функцию как параметр
  - создавать массивы указателей на функцию
  - вызывать функцию по указателю на нее

```
    fun1d g = f;
...
g(2.0);
```

# Функция как параметр функции

- Формальным параметром функции может быть функция
- Функция-формальный параметр объявляется:
  - непосредственно в заголовке:

```
<тип> <имя_функции>(<тип> (*<имя_указателя>)
  (<типы_параметров>)...)
```

  - пример:

```
int foo(int (*fun)(char*), double alpha);
```

# Функция как параметр функции

- Функция-формальный параметр объявляется:
  - с объявлением типа указателя:

```
typedef <тип> (*<новый_тип>)(<типы_параметров>);
<тип> <имя>(<новый тип> имя_функции,... )
```

  - пример:

```
typedef int (*fun_type)(char*);
int foo(fun_type fun, double alpha);
```

# Пример: решение нелинейного ур-я

```
typedef double (*fun1d)(double);
double Root(fun1d f, double x0, double eps)
{
  double dx  = eps*1.0e-4, df;
  double x = x0, fx = f(x);
  while ( fabs(fx) > eps )
  {
    df = (f(x + dx) - f(x - dx)) / (2.0 * dx);
    fx = f(x -= fx / df);
  }
  return x;
}
```

# Пример: решение нелинейного ур-я

```
double f1(double x)
{
  return sin(x) * cos(x);
}
double f2(double x)
{
  return exp(x) - 1.5;
}

int main()
{
   double r1 = Root(f1, 1.0, 1.e-5);
   double r2 = Root(f2, 0.0, 1.e-7);
}
```

# Массив указателей на функцию

```
int f2(int x);
int f3(int x);
int f4(int x);

typedef int (*fun_type)(int);

// Замыкание
int fun(int b, fun_type f)
{
  return f(b);
}
```

# Массив указателей на функцию

```
int main()
{
  int i, j, k, l, m, n;
  // Объявление массива указателей на функцию
  fun_type funs[3] = { &f2 };
  // Или: fun_type* funs = new fun_type[3];

  funs[1] =  f3;
  funs[2] = *f4;
  i = funs[0](2);
  j = funs[1](2);     // Вызов по указателю
  j = (*funs[1])(2);     // Вызов функции
  j = (*(funs+1))(2);    // Вызов указателя
  j = (**(funs+1))(2);      // Вызов функции
  k = funs[2](2);
  return 0;
}
```

# Задачи

- Разборка строки на слова
- Транспонирование матрицы
- Задачи для самостоятельной работы