

**Курс «Алгоритмы и алгоритмические языки»  
1 семестр 2016/2017**

**Лекция 8**

## Указатели

- ◇ `&` - операция адресации
- `*` - операция разыменования

```
int a = 1;
int *p;
p = &a;
*p = 2;
printf ("Значение переменной a = %d\n", *p);
printf ("Адрес переменной a = %p\n", p);
```

В результате выполнения фрагмента будет напечатано:

**Значение переменной a = 2**

**Адрес переменной a = 0xbffff7a4**

- ◇ `&foo` является константой, указатель – переменной `foo` должен быть *l*-значением (`lvalue`)
- ◇ Печать адреса – модификатор `%p`
- ◇ Нулевой указатель (никуда не указывающий) – `NULL` (константа в `stdlib.h`, может не иметь нулевого значения)

## Адресная арифметика

- ◆ В языке Си допустимы следующие операции над указателями:
  - ◆ сложение указателя с целым числом
  - ◆ вычитание целого числа из указателя
  - ◆ вычитание указателей
  - ◆ операции отношения и сравнения
- ◆ Пример. Пусть `sizeof (int) == 4`  
и пусть текущее значение `int* p1` равно 2012.  
После операции `p1++` значение `p1` будет 2016 (а не 2013),  
после операции `p1-3` – значение 2000.
  - ◆ при увеличении (уменьшении) на целое число `i` указатель будет перемещаться на `i` ячеек соответствующего типа в сторону увеличения (уменьшения) их адресов.

## Преобразование типа указателя

- ◆ Указатель можно преобразовать к другому типу, но такое преобразование типов обязательно должно быть явным. **Условие:** исходный указатель правильно *выравнен* для целевого типа. Значение указателя сохраняется.

Иногда такое преобразование типов может вызвать непредсказуемое поведение программы.

- ◆

```
#include <stdio.h>
int main (void)
{
    double x = 200.35, y;
    int *p;
    p = (int *)&x; /* &x ссылается на double,
                   а p имеет тип int* */
    y = *p; /* будет ли y присвоено
            значение 200.35? */
    printf ("значение x равно %f\n", x);
    printf ("значение y равно %f\n", y);
    return 0;
}
```

## Преобразование типа указателя

◆ Типичный вывод (GCC, Linux):

**значение x равно 200.350000**

**значение y равно 858993459.000000**

- ◆ В присваивании `y = *p;` загрузка `*p` считывает только первые **четыре** байта области памяти с адресом `&x` (т.к. `sizeof (int)` в данном случае равен 4)
- ◆ В представлении `200.35` в формате числа `double` первые четыре байта соответствуют целому числу **858993459**

◆ Таким образом, необходимо учитывать, что операции с указателями выполняются **в соответствии с базовым типом указателя.**

## Преобразование типа указателя

- ◇ Разрешено также преобразование целого в указатель и наоборот (поведение определяется реализацией). Однако пользоваться этим нужно очень осторожно.  
`aux = (void *) -1;`
- ◇ Допускается присваивание указателя типа `void *` указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Это позволяет использовать указатель типа `void *`, когда тип объекта неизвестен.
  - ◇ Использование типа `void *` в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа.
- ◇ Допускается неявное преобразование *менее* квалифицированного типа указателя к *более* квалифицированному (`int * → const int *`)
- ◇ Константа `NULL` неявно преобразуется к любому другому типу указателя

## ***Указатели и массивы***

- ◇ Указатель на первый элемент массива можно создать, присвоив переменной типа “указатель на тип элемента массива” имя массива без индекса:

```
int array[15];  
int *p, *q;  
p = array;  
q = &array[0];
```

- ◇ `p` и `q` указывают на начало массива `array[15]`

- ◇ Значение `array` изменить нельзя, а значение `p` – можно.

`array` не является *l*-значением, а `p` – является

`array = p; array++` – писать нельзя (это ошибки)

`p = array; p++` – писать можно (и нужно)

## ***Указатели и массивы***

### ◆ Индексирование указателей

```
int *p, a[10]; /* два способа присвоить 100 */
               /* 6-ому элементу массива a[10] */

p = a;
*(p + 5) = 100; /* адресная арифметика */
p[5] = 100; /* индексирование указателя */
```

### ◆ Сравнение указателей

Если  $p$  и  $q$  являются указателями на элементы одного и того же массива и  $p < q$ , то:

$q - p + 1$  равно количеству элементов массива от  $p$  до  $q$  включительно.

Можно написать:

```
if (p < q)
    printf ("p ссылается на меньший адрес, чем q");
```



## Массивы указателей

- Указатели могут быть собраны в массив:

```
int *mu[27]; /* это массив из 27 указателей на int */
int (*um)[27]; /* это указатель на массив из 27 int */
```

- Пример

```
static void error (int errno)
{
    static char *errmsg[] = {
        "переменная уже существует",
        "нет такой переменной",
        <...>
        "нужно использовать переменную-указатель"
    };
    printf ("Ошибка: %s\n", errmsg[errno]);
}
```

- Имя массива указателей – пример многоуровневого указателя. Массив `errmsg` можно представить как `char **errmsg`

## Функции

### ◆ **Объявление функции:**

*тип\_возвр\_значения имя\_функции(тип параметр,  
тип параметр, ..., тип параметр);*

`int atoi (char s[]);`

`void QuickSort (char *items, int count);`

◆ Тип возвращаемого значения `void` означает, что функция не возвращает значения.

### ◆ **Определение функции:**

*объявление\_функции {тело\_функции}*

◆ **Областью действия** функции является весь программный файл, в котором она объявлена, начиная со строки, содержащей ее объявление.

◆ Если в программном файле вызывается какая-либо функция, она *обязательно должна быть объявлена в этом программном файле до ее вызова.*

◆ Директива препроцессора `#include <имя_библиотеки.h>` вставляет в программу объявления всех функций соответствующей библиотеки

## **Вызов функции**

- ◇ Если функция  $f( )$  возвращает значение типа *тип*, то вызов этой функции может иметь вид:  
$$v = f( );$$
где  $v$  – переменная типа *тип*.
  
- ◇ Если функция  $f( параметр )$  не возвращает значений, вызов этой функции имеет вид:  
$$f( аргумент );$$
  
- ◇ **В языке Си все аргументы передаются по значению** (т.е. передаются только значения аргументов, и эти значения копируются в локальную область памяти функции).
  
- ◇ Если аргументом является указатель, его значением может быть адрес объекта вызывающей функции, что обеспечивает вызываемой функции доступ к объекту.

## *Указатели и аргументы функций*

- ◇ Используя аргументы-указатели, функция может обращаться к объектам вызвавшей ее функции.
- ◇ Использование указателей позволяет избежать копирования сложных структур данных: вместо этого передаются *указатели* на эти структуры.
- ◇ **Пример.** Функция `void swap(int x, int y);` меняет местами значения переменных `x` и `y`.

*Неправильный вариант:*

```
void swap (int x, int y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

*Правильный вариант:*

```
void swap (int *px, int *py)
{
    int tmp;
    tmp = *px;
    *px = *py;
    *py = tmp;
}
```

## Вызов функции

- ◇ Массив всегда передается с помощью указателя на его первый элемент.

```
int asum1d (int a[], int n) {  
    int s = 0;  
    for (int i = 0; i < n; i++)  
        s += a[i];  
    return s;  
}
```

Можно объявить массив `a` в списке параметров как `const a[]`.

- ◇ Функции с переменным числом параметров:

```
int scanf (const char *, ...);
```

Всегда должен быть явно задан хотя бы один параметр.

После многоточия не должно быть других явных параметров.

Обработка переменных параметров – файл `stdarg.h`,  
макросы `va_start/va_arg/va_end`.