Курс «Алгоритмы и алгоритмические языки»
1 семестр 2014/2015

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.

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 память 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**.

Функции

- Пример:

```
#include <ctype.h>
int atoi (char *s)
{
  int n, sign;
  for (; isspace (*s); s++)
    ;
  sign = (*s == '-') ? -1 : 1;
  if (*s == '+' || *s == '-')
    s++;
  for (n = 0; isdigit (*s); s++)
    n = 10 * (*s - '0');
  return sign * n;
}
```

- Стандартная библиотека **`ctype`** содержит такие функции, как **`isspace()`** , **`isdigit()`** и др.

Возврат из функции

- Возврат из функции в точку вызвавшей ее функции, следующей за точкой вызова функции, осуществляется:
  - либо при выполнении оператора **`return`**,
  - либо после выполнения последнего оператора функции, если она не содержит оператора **`return`**.

```
#include <string.h>
#include <stdio.h>
void print_str_reverse (char *s)
{
  register int i;
  for (i = strlen (s) – 1; i >= 0; i--)
    putchar (s[i]);
}
```

- Если тип функции не **`void`**, то в ее теле должен быть хотя бы один оператор **`return`** с возвращаемым значением
- Если у функции несколько операторов **`return`**, возврат осуществляется **немедленно** по тому из них, который будет выполнен первым.

Результат выполнения функции

- Все функции, кроме тех, которые относятся к типу **`void`**, возвращают значение, которое определяется выражением в операторе **`return`**.
- Помимо вычисления возвращаемого значения, функция может изменять значения переменных вызывающей функции (по указателю), а также изменять значения глобальных переменных.
- Результаты вызова функции, не связанные непосредственно с вычислением возвращаемых значений, составляют *побочный эффект* функции.
- Выделяют следующие виды функций:
  (1) Функции, которые выполняют операции над своими аргументами с единственной целью – вычислить возвращаемое значение.
  (2) Функции, которые обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка.
  (3) Функции, возвращающие несколько значений (через указатели-аргументы и через возвращаемое значение).
  (4) Функции, не возвращающие значений. Все такие функции имеют тип **`void`**.

Результат выполнения функции

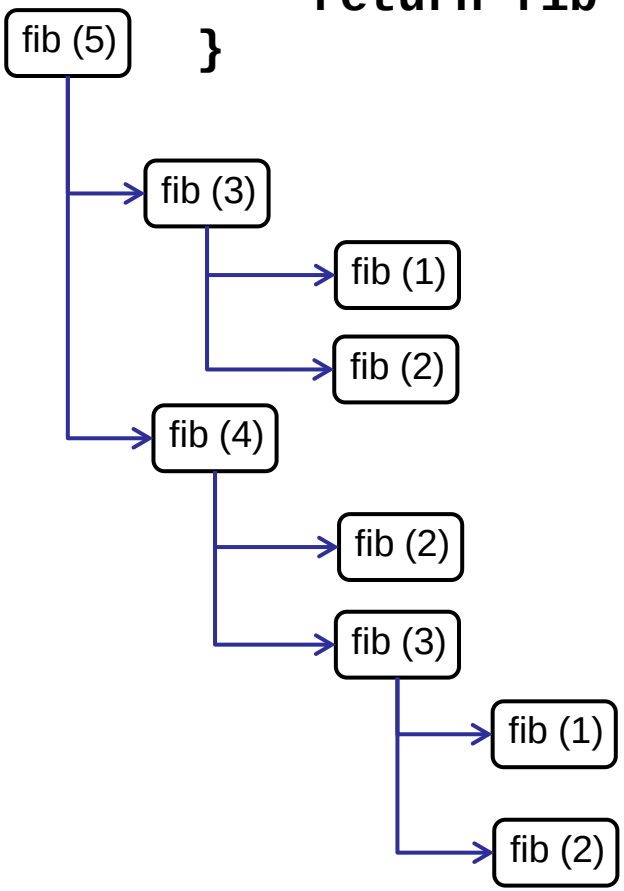
- Возвращаемым значением может быть указатель. Требуется, чтобы в объявлении такой функции тип возвращаемого указателя был объявлен точно: нельзя объявлять возвращаемый тип как **`int *`**, если функция возвращает указатель типа **`char *`**.
- Пример функции, возвращающей указатель (поиск первого вхождения символа **`c`** в строку **`s`**):

```
char *match (char c, char *s)
{
  while (c != *s && *s)
    s++;
  return s;
}
```

Рекурсия

- В языке Си функция может быть *рекурсивной*, т.е. вызывать саму себя:

```
int fib (int n)
{
  if (n == 1 || n == 2)
    return 1;
  else
    return fib (n – 2) + fib (n – 1);
}
```



Рекурсия

- Рекурсивные функции часто неэффективны по сравнению с их нерекурсивными вариантами:

```
int fibn (int n) {
  int i, g, h, fb;
  if (n == 1 || n == 2)
    return 1;
  else
    for (i = 2, g = h = 1; i < n; i++) {
      fb = g + h;
      h = g;
      g = fb;
    }
  return fb;
}
```

- Функция **`fib`** работает за экспоненциальное время и линейную память,
функция **`fibn`** – за линейное время и константную память.

*Хвостовая рекурсия**

- *Хвостовая рекурсия* (tail recursion) – рекурсивный вызов в самом конце функции. Как правило, этот вызов может быть оптимизирован компилятором в цикл.

```
int fact (int n) {
  if (n == 0)
    return 1;
  else
    return n*fact (n-1);
}
```

```
int fact (int n) {
  return tfact (n, 1);
}
int tfact (int n, int acc) {
  if (n == 0)
    return acc;
  return tfact (n-1, n*acc);
}
```

```
int fact (int n) {
  int t_n = n, t_acc = 1;

/* tfact встроена в fact и оптимизирована в цикл */
start:
  if (t_n == 0)
    return t_acc;
  t_acc = t_n * t_acc;
  t_n = t_n – 1;
  goto start;
}
```

Ключевое слово `inline`: встраиваемые функции (C99)

```
#include <stdio.h>
inline static int max (int a, int b)
{
  return a > b ? a : b;
}
int main (void)
{
  int x = 5, y = 17;
  printf ("Наибольшим из чисел %d и %d является %d\n",
           x, y, max (x, y));
  return 0;
}
```

◈ При обычной реализации **`inline`** приведенная программа эквивалентна:

```
#include <stdio.h>
inline static int max (int a, int b)
{
  return a > b ? a : b;
}
int main (void)
{
  int x = 5, y = 17;
  printf ("Наибольшим из чисел %d и %d является %d\n",
           x, y, (x > y ? x : y));
  return 0;
}
```

Указатели на функцию

- Каждая функция располагается в памяти по определенному адресу. Адресом функции является ее точка входа (при вызове функции управление передается именно на эту точку).
- Присвоив значение адреса функции переменной типа указатель, получим указатель на функцию.
- Указатель функции можно использовать вместо ее имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим функциям в качестве их аргумента.
- Имя функции **`f()`** без скобок и аргументов (**`f`**) по определению является указателем на функцию **`f()`** (аналогия с массивом).

```
int (*pf) (const char*, const char*);
char *s1, *s2;
int x = (*pf) (s1, s2);
int y = pf (s2, "string constant");
```

Указатели на функцию

- ***Пример***. Сравнение двух строк символов, введенных пользователем (функция **check()**).

```
#include <stdio.h>
#include <string.h>

static void check (char *a, char *b,
                   int (*pf) (const char*, const char*)) {
  printf ("Проверка на совпадение: ");
  if (! pf (a, b))
    printf ("равны\n");
  else
    printf ("не равны\n");
}

int main (void) {
  char s1[80], s2[80];

  printf ("Введите две строки \n");
  fgets (s1, sizeof (s1), stdin); s1[strlen (s1) - 1] = 0;
  fgets (s2, sizeof (s2), stdin); s2[strlen (s2) - 1] = 0;
  check (s1, s2, strcmp);
  return 0;
}
```

- Объявление **`int (*p)(const char *, const char *);`** сообщает компилятору, что **`p`** – указатель на функцию, имеющую два параметра типа **`const char *`** и возвращающую значение типа **`int`**.
- Скобки вокруг **`*p`** нужны, так как операция **`*`** имеет более низкий приоритет, чем **`()`**: если написать **`int *p(...)`**, получится, что объявлен не указатель на функцию, а функция **`p`**, которая возвращает указатель на целое.
- **`(*cmp)(a, b)`** эквивалентно **`cmp(a, b)`.**
- У функции **`check`** три параметра: два указателя на тип **`char`** и указатель на функцию **`pf`**. Указатель **`pf`** и функция **`strcmp`** имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру **`pf`**.
- В данном случае использование указателя на функцию позволяет не менять программу сравнения, и тем самым получается более общий алгоритм.

```
int compvalues (const char *a, const char *b) {
  return atoi (a) != atoi (b);
}
```

- Массивы указателей на функцию: гибкая обработка событий