Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 11

12 октября 2019 г.

Пример: перевод строковой записи числа в целый тип

```c
#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;
}
```

Возврат из функции

Возврат из функции в точку вызвавшей её 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 её теле на каждом пути выполнения должен быть 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.

Результат выполнения функции

Выделяют следующие виды функций:

- выполняют операции над своими аргументами с единственной целью — вычислить возвращаемое значение;
- обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка;
- возвращающие несколько значений (через указатели-аргументы и через возвращаемое значение);
- 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 *`.

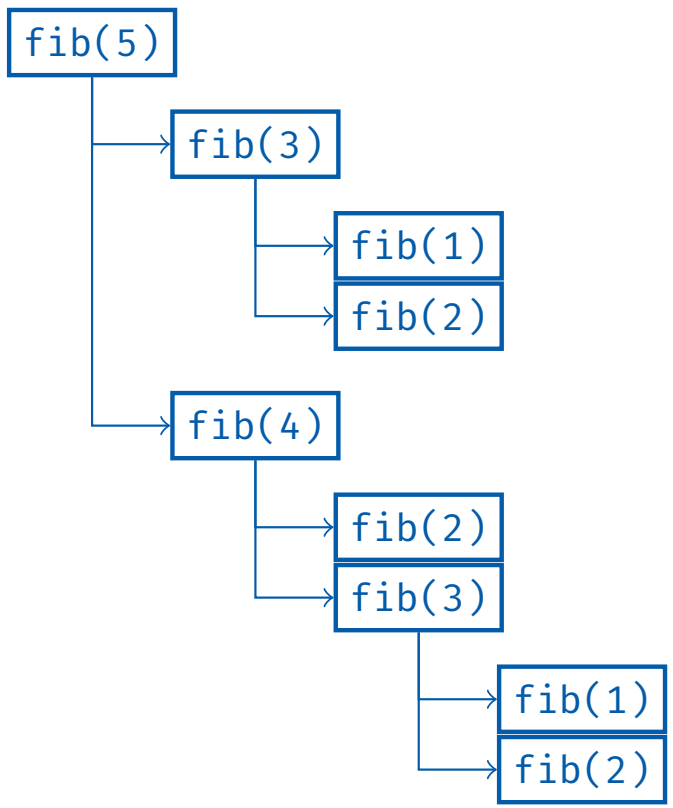
Пример функции, возвращающей указатель: поиск первого вхождения символа `c` в строку `s`.

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

Рекурсия

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);
}
```

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) {
  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;
}
```

```
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);
}
```

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 ("Maximum␣of␣%d␣and␣%d␣is␣%d\n",
          x, y, max (x, y));
  return 0;
}
```

Ключевое слово `inline`: встраиваемые функции (C99)

При типичной реализации `inline` программа будет преобразована как

```
#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 ("Maximum␣of␣%d␣and␣%d␣is␣%d\n",
          x, y, (x > y ? x : y));
  return 0;
}
```

Указатели на функцию

Каждая функция располагается в памяти по определенному адресу. Адресом функции является её 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 её 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 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

Каждая функция располагается в памяти по определенному адресу. Адресом функции является её 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 её 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 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

```
#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;
}
```

Указатели на функцию. Пример

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)`.

Указатель `pf` и функция `strcmp` имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру `pf`.

Указатели на функцию. Пример

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.