Указатели (продолжение), stdarg

А. Г. Фенстер, fenster@fenster.name

27 февраля 2010 г.

1 Указатели на функции

В языке C возможно определить переменную типа «указатель на функцию», присвоить ей какое-либо значение (адрес ранее определённой функции) и затем использовать её как функцию. Обратите внимание на скобки, из-за которых две следующих строчки имеют совершенно различный смысл:

```
int *f(char);   /* объявление функции f, принимающей char и возвращающей int *       */
int (*g)(char); /* g -- указатель на функцию, принимающую char и возвращающую int    */
```

Указателю на функцию можно присвоить значение. Если существует функция `f1`, имеющая набор параметров нужных типов и возвращающая значение указанного типа:

```
int f1(char c)
{
    ...
}
```

мы можем присвоить значение переменной `g`:

```
g = &f1;   /* оператор взятия адреса */
g = f1;    /* обычно допустима и такая запись */
```

После такого присваивания можно записать вызов функции:

```
x = (*g)('a');   /* скобки обязательны! */
x = g('a');        /* обычно можно и так */
```

Приведём несколько примеров использования указателей на функции.

1.1 Таблица значений функции

Простой пример: пусть функция `print` печатает значения произвольной функции на отрезке от `a` до `b` с шагом `s`:

```
void print(double (*f)(double), double a, double b, double s)
{
    for ( ; a <= b; a += s)
    {
        printf("%f\t%f\n", a, f(a));
    }
}
```

Написав такую функцию, можно напечатать значения любой подходящей функции:

```
#include <math.h>

print(sin, 0, 1, 0.1);
print(cos, 0, 1, 0.1);
```

Если бы не было возможности объявить указатель на функцию, в функции `print` пришлось бы перечислять все допустимые функции при помощи `if` или `switch`, что было бы менее удобно.

1.2 Сортировка и поиск

Функция `print` в предыдущем примере является простым примером функции, использующей технику *обратного вызова* (callback): она вызывает функцию, определённую пользователем. Более полезными примерами таких функций являются стандартные функции поиска и сортировки из `stdlib.h`:

```
/* быстрая сортировка */
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));

/* бинарный поиск */
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Обе этих функции принимают последним параметром указатель на функцию, сравнивающую два произвольных объекта, и выполняют, соответственно, быструю сортировку массива `base`, состоящего из `nmemb`

элементов размером size каждый, и бинарный поиск элемента с ключом key в упорядоченном массиве, используя переданную функцию для сравнения элементов. Эта функция должна возвращать 0, если элементы равны; положительное число, если первый аргумент больше второго; отрицательное, если меньше.

Для примера отсортируем массив целых чисел:

```
int intcmp(const void *p1, const void *p2)
{
    return (*(int *)p1 - *(int *)p2);
}

int A[N] = { ... };
qsort(A, N, sizeof(int), intcmp);
```

Функция intcmp принимает два указателя на элементы массива в виде указателей на произвольный тип void *. Она предполагает, что по указанным адресам расположены значения типа int, и т. к. разыменовать void * напрямую нельзя, сначала приводит оба указателя к типу int *, а затем сразу же разыменовывает их. Разность сравниваемых чисел как раз даёт нужный результат (0 при равенстве и положительное или отрицательное число в других случаях).

Аналогичным образом работает функция bsearch; она возвращает адрес найденного элемента или NULL, если элемент найден не был.

При помощи функции qsort удобно сортировать массивы, состоящие из структур любой сложности, необходимо лишь предоставить функцию, умеющую сравнивать элементы такого типа.

1.3 Перебор элементов хранилища

Функции обратного вызова можно использовать при переборе элементов некоторого хранилища (контейнера) — например, списка. Пусть определёна структура элемента списка struct item:

```
struct item
{
    int data;
    struct item *next;
};
```

Чтобы скрыть от пользователя реализацию списка, мы можем написать функцию, перебирающую его элементы и вызывающую для каждого из них указанную пользователем функцию:

```
typedef struct item *list;

void foreach(list head, void (*func)(int))
{
    struct item *p;
    for (p = head; p; p = p->next)
    {
        func(p->data);
    }
}
```

В таком случае пользователь сможет, например, напечатать значения элементов списка, не зная самой структуры списка:

```
void print_element(int x)
{
    printf("%d\n", x);
}

list head;
...
foreach(head, print_element);
```

при этом вся работа со списком от пользователя скрыта в функции `foreach`. В функциональных языках программирования такую функцию обычно называют `map`.

Примечание. Про списки мы подробно поговорим на одном из следующих занятий, если этот пример сейчас не очень понятен — вернитесь к нему позже.

2 Функции с переменным количеством параметров

Как вы, вероятно, уже заметили, некоторые функции в C (например, `printf` и `scanf`) могут принимать различное количество аргументов. Оказывается, достаточно несложно сделать свою функцию, которая будет действовать таким же образом. Функция, принимающая переменное количество параметров, объявляется с указанием многоточия в конце списка параметров, например, так:

```
int sum(int count, ...)
{
    ...
}
```

Функция с переменным количеством параметров обязана иметь хотя бы один «обычный» (именованный) параметр. Это связано с тем, что

адреса следующих параметров обычно вычисляются на основе адреса последнего из именованных параметров.

Эти действия выполняют специальные макросы из `stdarg.h`:

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Обратите внимание: это не функции, а именно макросы (определённые с помощью `#define`). Реализовать такую функциональность при помощи функций было бы невозможно (тип параметра `last` у `va_start` не определён, `va_arg` принимает вторым параметром имя типа и возвращает значение такого типа — всё это магия, функциям недоступная).

Используются эти макросы следующим образом. Функция, которая хочет использовать параметры, переданные через многоточие, заводит локальную переменную типа `va_list` и вызывает макрос `va_start`, передавая в качестве параметра `last` **последний** из именованных параметров. После этого последовательные применения макроса `va_arg` позволяют получить очередные параметры функции, приведённые к указанному типу. В конце необходимо вызвать макрос `va_end`.

Важно: при вызове функции с переменным количеством параметров параметры типа `char` преобразуются к `int`, параметры типа `float` — к `double`.

Реализуем функцию, вычисляющую сумму своих параметров. Первым параметром она будет получать количество параметров.

```
int sum(int count, ...)
{
    va_list vl;
    int result = 0;
    va_start(vl, count);
    for ( ; count > 0; count--)
    {
        result += va_arg(vl, int);
    }
    va_end(vl);
    return result;
}
```

Использование этой функции:

```
int x, y;

x = sum(5, 1, 2, 3, 4, 5); /* сумма пяти чисел */
y = sum(3, 1, 2, 3);       /* сумма трёх чисел */
```

Основной проблемой является передача функции информации о том, сколько параметров считывать (и какого они типа). Различные функции получают эту информацию по-разному: передачей количества параметров отдельным аргументом (как в примере выше), ограничением списка параметров нулём (как в функциях типа `execl` в UNIX) или, например, при помощи форматной строки (как `printf`/`scanf`).

Пример функции, рассматривающей параметры до тех пор, пока не встретится ноль:

```
int sum(int first, ...)
{
    va_list vl;
    int result = first;
    va_start(vl, first);
    while (first != 0)
    {
     first = va_arg(vl, int);
        result += first;
    }
    va_end(vl);
    return result;
}

...
int x = sum(1, 2, 3, 4, 5, 0);
```

3 Реализация функции `printf`

Реализуем функцию, подобную `printf`. «Настоящий» `printf` возвращает количество напечатанных символов; для простоты наша функция вместо этого не будет ничего возвращать. Пусть функция `print_integer` печатает (при помощи одного или нескольких вызовов `putchar`) число, данное ей в качестве параметра. Реализовать её можно, к примеру, так (простой вариант с вычислением максимальной степени числа 10, не превосходящей данного числа):

```
void print_integer(int x)
{
    int power = 1;
    if (x < 0)
    {
        x = -x;
        putchar('-');
    }

    while (x / power >= 10)
    {
```

```
        power *= 10;
    }

    while (power != 0)
    {
        putchar((x / power) % 10 + '0');
        power /= 10;
    }
}
```

Любителям функционального подхода, возможно, понравится другая реализация, в которой не требуется вычислять степень. Если бы не отдельная проверка на число 0, то функция была бы ещё короче:

```
void print_integer(int x)
{
    int power = 1;
    if (x < 0)
    {
        putchar('-');
        print_integer(-x);
        return;
    }

    if (x == 0)
    {
        putchar('0');
        return;
    }

    if (x >= 10)
    {
        print_integer(x / 10);
    }

    putchar(x % 10 + '0');
}
```

Обе этих реализации работают правильно для всех чисел типа `int`, кроме одного.

Упражнение. Что это за число?

Используя для печати целого числа реализованную выше функцию `print_integer`, для печати символа — `putchar`, для печати строки — `fputs` (не `puts`, поскольку он добавляет в конец `\n`), напишем свой мини-вариант `printf`:

```
void myprintf(char *fmt, ...)
{
    va_list vl;
    int d;
    char c, *s;

    va_start(vl, fmt);
```

```
    while (*fmt)  /* *fmt -- текущий символ строки */
    {
        if ('%' == *fmt)
        {
            fmt++;  /* сдвиг */
            switch (*fmt) {
            case 's':
                s = va_arg(vl, char *);
                fputs(s, stdout);
                break;
            case 'd':
                d = va_arg(vl, int);
                print_integer(d); /* печать int */
                break;
            case 'c':
                c = (char)va_arg(vl, int);  /* int, не char! */
                putchar(c);
            default:
                putchar(*fmt);
            }
         }
         else
         {
             putchar(*fmt);
         }
         fmt++;    /* сдвиг */
    }
    va_end(vl);
}
```

Полезные функции, принимающие `va_list`:

```
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
```

Эти функции удобно использовать при реализации своей функции, являющейся надстройкой над `printf`/`fprintf` (например, функции для удобной печати отладочной информации):

```
void debug(char *fmt, ...)
{
    va_list vl;
    va_start(vl, fmt);
    printf("DEBUG: ");
    vprintf(fmt, vl);
    va_end(vl);
}

...
debug("line %d: x = %d, y = %d\n", __LINE__, x, y); /* __LINE__ -- стандартный макрос */
```