Посимвольный ввод/вывод, массивы

А. Г. Фенстер, `fenster@fenster.name`

19 февраля 2010 г.

Конспект семинара №2 по программированию для студентов 1 курса ММФ НГУ. Помните, что чтение конспекта не делает посещение соответствующего семинара необязательным! О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу `fenster@fenster.name`. Также буду рад получить любые комментарии по поводу этого текста.

1 Повторение

Задача. Для нахождения наибольшего общего делителя двух натуральных чисел можно использовать *алгоритм Евклида*:

$$\text{НОД}\,(a, b) = \begin{cases} b, \text{ если } a \texttt{ \% } b = 0; \\ \text{НОД}\,(b,\ a \texttt{ \% } b) \text{ в противном случае,} \end{cases}$$

Реализуйте его при помощи рекурсивной функции.

Как и в примере с факториалом, рекурсия здесь является хвостовой. На практических занятиях в терминальном классе вторая задача — реализация алгоритма Евклида при помощи цикла.

2 Посимвольный ввод/вывод

В языке C для обозначения символов (букв, цифр, знаков препинания) используются апострофы: `'A'`, `'1'`, `'\n'` (символ перевода строки). Пробел, который обозначается `' '`, для удобства чтения мы будем писать как `'␣'`. Не путайте апострофы с двойными кавычками, которые ограничивают строки; про них будет рассказано позже.

Каждому символу в компьютере соответствует некоторое число — *код символа*. Например, заглавная латинская буква `'A'` обычно имеет код 65. В языке C символ (в апострофах) и его код взаимозаменяемы: везде, где в программе встречается `'A'`, можно написать 65, и наоборот. Конечно, явное указание символа предпочтительнее: код

```
char c = 'A';
```

всегда выглядит понятнее, чем эквивалентный ему

```
char c = 65;
```

поэтому следует придерживаться первого варианта.

Не следует путать символы `'0'`, `'1'`, ..., `'9'` с числами 0, 1, ..., 9. Символу `'0'` соответствует код 48, символу `'1'` — 49 и т. д.. То есть, переменные `a` и `b` в следующем примере **не равны**:

```
char a = '0'; /* a == 48 */
char b =  0;  /* b == 0  */
```

Для ввода одного символа с клавиатуры (из потока стандартного ввода) может использоваться функция `getchar`, не принимающая параметров и возвращающая `int`. Почему `int`, а не `char`? Дело в том, что `getchar` может вернуть любой из возможных $2^8 = 256$ символов (весь диапазон типа `char`), но кроме этого необходимо иметь возможность сигнализировать об ошибке (например, конец потока ввода). Таким образом, вызов `getchar()` возвращает код прочитанного символа либо константу `EOF`, которая не совпадает ни с каким из кодов символов (она обычно равна -1).

Необходимо отметить, что в реальности запущенная программа, в которой есть вызов `getchar()`, скорее всего будет ожидать нажатия клавиши Enter (перевода строки). Это поведение определяется настройкой терминала. Если в программе выполняются три последовательных вызова `getchar()`, а пользователь ввёл `AB` и нажал клавишу Enter, то первый вызов вернёт символ `'A'` (причём только в момент нажатия Enter), второй вызов сразу же после этого вернёт `'B'`, а третий — `'\n'`.

Функция `putchar` принимает параметром число и печатает на экран (в поток стандартного вывода) символ с таким кодом. Фактически, вызов `putchar(x)` эквивалентен `printf("%c", x)`. Заметьте, что это не то же самое, что `printf("%d", x)` — последний вызов `printf` напечатает не символ, хранящийся в переменной `x`, а соответствующее ему целое число — т. е. его код.

Прежде чем двигаться дальше, желательно разобраться, что представляет собой код символа (в частности, как хранятся цифры, латинские и русские буквы). Об этом — следующее «лирическое отступление».

2.1 Грустная история русских кодировок

Под *кодировкой символов* обычно понимают соответствие между используемыми в компьютере символами (буквами, цифрами, знаками препинания и т. п.) и их числовыми кодами. Все используемые на настоящий момент кодировки так или иначе произошли от кодировки ASCII (American Standard Code for Information Interchange), определяющей коды цифр, латинских букв и некоторых спецсимволов (см. таблицу ниже). К сожалению, русских букв (кириллицы) в этом наборе не наблюдается.

Коды	**Символы**
0–31	Специальные и управляющие символы; 10 – `'\n'`
32	Пробел `'␣'`
33–47	Знаки пунктуации
48–57	Цифры от `'0'` до `'9'` (подряд)
58–64	Знаки пунктуации
65–90	Заглавные латинские буквы от `'A'` до `'Z'`
91–96	Знаки пунктуации
97–122	Строчные латинские буквы от `'a'` до `'z'`
123–126	Знаки пунктуации
127	Спецсимвол

Т. к. ASCII определяет только первые 128 кодов, а один байт может принимать 256 (2^8) различных значений, оставшиеся 128 кодов могут быть определены по-разному. Например, кодировка CP437 (CP — от codepage) помещает в «свободные» места элементы псевдографики (одинарные и двойные линии и их различные пересечения), акцентированные символы (ä, ú и т. п.), некоторые буквы греческого алфавита.

Попытки русифицировать кодировку ASCII породили настоящий хаос, который закончится, вероятно, только после повсеместного перехода на многобайтовую кодировку UTF-8 (всё к этому идёт, но настанет этот счастливый момент не скоро). В UTF-8 один символ может занимать несколько (от одного до четырёх) байт, при этом символы, определённые в наборе ASCII, имеют такой же код, как и в ASCII.

Что касается однобайтовых кодировок с поддержкой русских букв — все они строятся по одному принципу: первые 128 символов (коды от 0 до 127 включительно) берутся из набора ASCII, а следующие 128 (коды от 128 до 255) заполняются более-менее произвольно. Если рассматри-

вать только дожившие до настоящего момента кодировки, то в нашем распоряжении имеются:

- Кодировка CP866 («кодировка DOS»), которая была бы полностью (и вполне заслуженно) забытой, если бы не окошки `cmd.exe` в ОС Windows. Эта кодировка отличалась тем, что между строчными буквами `'п'` и `'р'` была оставлена «дырка» из 48 кодов, которые в CP437 соответствовали символам псевдографики. Это позволяло запускать программы, написанные для CP437, на компьютерах с CP866, при этом псевдографические «окошки» оставались таковыми (вспомните, например, Turbo Pascal 7.0).

- Кодировка KOI8-R (КОИ — код обмена информацией) заполняет последние 128 позиций русскими буквами таким образом, что если отбросить старший бит кода (разделить нацело код символа на 2), получится более-менее соответствующая латинская буква (я намеренно не говорю ничего про «промежуточную» кодировку KOI7, желающие могут прочитать о ней в Интернете). Это соответствие очень удобно в те редкие моменты, когда ваш терминал «забывает» о том, что он восьми-, а не семибитный, но приводит к тому, что русские буквы в таблице идут не в алфавитном порядке, а примерно так: `'ю'`, `'а'`, `'б'`, `'ц'`, `'д'`, ... Неудобства такого подхода вскоре станут вам понятными. По историческим причинам именно KOI8-R распространена в UNIX-подобных операционных системах.

- Кодировка CP1251, она же Windows-1251, используется в одноимённой операционной системе. Она размещает русские буквы по алфавиту и без «дырок». К сожалению, букве `'я'` достался последний код 255, а число 255, будучи приведённым к знаковому типу `char`, даст -1. Как вы помните, именно минус единице равна константа `EOF`, следовательно, если программист под Windows неаккуратно работает с типами данных, то введённая по запросу программы буква `'я'` иногда может привести к завершению чтения.

Как видите, с однобайтовыми кодировками счастья практически нет. Стоит ещё заметить, что если вы хотите, чтобы код русских букв представлялся *положительным* числом, используйте для их хранения вместо `char` тип `unsigned char`. На этом «лирическое отступление» про кодировки, думаю, следует закончить.

3 Преобразование символов-цифр в числа

Если известно, что переменная `char c` содержит цифру, т. е. один из символов `'0'`, `'1'`, ..., `'9'`, то для вычисления соответствующего ему числа (0, 1, ..., 9) не нужно писать условную инструкцию `if` или инструкцию выбора `switch`. Достаточно вспомнить, что цифры в таблице ASCII идут по порядку, так что искомым числом будет являться разность символа `c` и символа `'0'`:

...	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	...
...	48	49	50	51	52	53	54	55	56	57	...

```
char c = '7';        /* c == 55 */
int value = c - '0'; /* value == 55 - 48 == 7 */
```

Задача. Реализуем функцию, читающую с клавиатуры (из стандартного ввода) натуральное число. Эта функция не будет принимать аргументов; чтобы показать это, вместо списка параметров функции пишется слово `void`. Возвращает функция прочитанное число. В переменной `result` будем накапливать результат:

```
int read_integer(void)
{
    int result = 0;
    ...
    return result;
}
```

Идея реализации: заметим, что если введено число 123 и мы уже успели при помощи двух вызовов `getchar()` прочитать `'1'` и `'2'` (соответственно, в переменной `result` записано 12), то для получения 123 из 12 необходимо старое значение (12) умножить на 10 и добавить значение прочитанной цифры (3). Соответственно, необходимо выполнять `getchar()` в цикле, пока читаются цифры, и на каждом шаге вычислять новое значение переменной `result`.

Для проверки того, что в переменной `c` хранится цифра, можно либо выполнить явное сравнение:

```
if (c >= '0' && c <= '9')
    ...
```

либо воспользоваться функцией `isdigit` из `<ctype.h>`:

```
#include <ctype.h>

...
if (isdigit(c))
    ...
```

4 Разворот ввода

Задача. Напишем функцию, которая читает с клавиатуры символы, пока не будет нажата клавиша Enter, и печатает их в обратном порядке. Обратите внимание, что количество этих символов заранее неизвестно.

Задача может быть решена без использования массивов (про которые мы ещё не говорили). Единственным доступным нам на текущий момент способом хранения неограниченного (в разумных пределах) количества символов является рекурсия: ведь при рекурсивном вызове (как и при любом вызове функции) переменные вызывающей функции остаются в памяти и их значения становятся доступными после завершения рекурсивного вызова.

Наша функция не будет ни принимать, ни возвращать параметров (вместо типа возвращаемого значения напишем слово `void`). В других языках (например, в Pascal) функции, не возвращающие значения, называются *процедурами*, мы изредка будем использовать это слово в таком же значении. Таким образом, заголовок нашей функции будет таким:

```
void reverse(void)
{
    ...
}
```

Первое, о чём нужно подумать, придумывая рекурсивную функцию — о том, как и когда рекурсия будет завершаться, т. е. определить её *терминальную ветвь*. В нашем случае функция не должна ничего делать, если введён символ перевода строки:

```
    int x = getchar();
    if ('\n' == x)
        return;
```

Мы научились разворачивать пустые строки: терминальная ветвь рекурсии — это база индукции; простой случай, для которого действие функции очевидно.

Пусть введён не перевод строки, а какой-либо другой символ `x`. Сведём задачу к задаче меньшего размера: развернуть строку, начинающуюся с символа `'c'` — это всё равно что развернуть и напечатать все оставшиеся до нажатия Enter символы, а затем напечатать символ `'c'`:

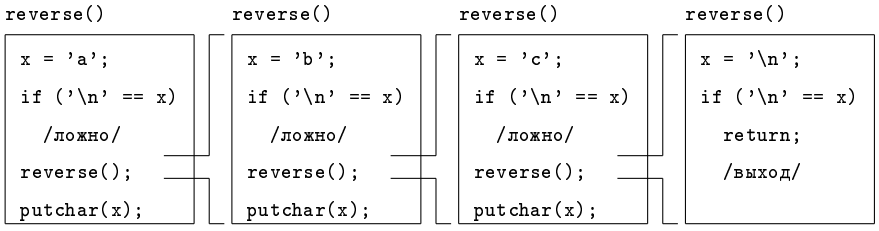
$$\text{rev}(\texttt{"coffee"}) = \text{rev}(\texttt{"offee"})\ \ \texttt{'c'}.$$

Прежде чем читать дальше, пожалуйста, попробуйте написать функцию `reverse` самостоятельно!

Осталось понять, что «развернуть и напечатать все оставшиеся символы» — это просто рекурсивный вызов функции `reverse`, и у нас получится такой код:

```
void reverse(void)
{
    int x = getchar();
    if ('\n' == x)
        return;
    reverse();
    putchar(x);
}
```

Нарисуем последовательность вызовов функции `reverse`, если пользователь ввёл `abc` Enter:



Когда последний из вызовов `reverse` завершается, управление переходит к предыдущей функции `reverse` (в которой переменная `x` равна `'c'`), на следующую после рекурсивного вызова инструкцию. Эта инструкция — `putchar`, который печатает `c`. Затем и этот вызов `reverse` завершается, управление переходит к предыдущей функции (в которой `x` равен `'b'`), он печатается, и так далее до тех пор, пока все вызовы функции `reverse` не завершатся.

Заметьте, что эта рекурсия не является хвостовой: рекурсивный вызов не является последней инструкцией в функции. Избавиться от рекурсии, переписав эту функцию в виде цикла без использования дополнительной памяти, нельзя.

5 Массивы

Массивом на семинарах мы будем называть последовательность пронумерованных (индексированных) переменных одного и того же типа. На лекциях будет дано более формальное определение. При объявлении массива указывается тип и количество элементов следующим образом:

```
int A[5];
```

Элементы нумеруются от 0, так что этот массив будет состоять из следующих пяти элементов: `A[0]`, `A[1]`, `A[2]`, `A[3]`, `A[4]`. Каждый элемент – отдельная переменная типа `int`.

Массивы могут быть как одномерными (предыдущий пример), так и многомерными. Например, вот двумерный массив, который можно представить и как линейную структуру, и как матрицу:

```
int A[2][3];
```

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]

Для работы с массивами обычно используются циклы. Например, вот так можно напечатать все элементы массива `A`:

```
int i;

for (i = 0; i < 5; i++)
    printf("%d ", A[i]);
```

При создании (и **только** при создании) массива его элементы можно инициализировать:

```
int A[5] = { 1, 2, 3, 4, 5 };
```

Если массив инициализируется, его размер можно не указывать: он будет вычислен автоматически. Если же, напротив, размер указан, а в инициализаторе не хватает элементов, остаток массива будет заполнен нулями — но только если инициализатор присутствует, в противном случае значения элементов будут не определены (если массив создан внутри функции).

Пример инициализации массива нулями при создании:

```
int A[5] = { 0 }; /* неполный инициализатор: остальные элементы равны нулю */
```

Чтобы при изменении размера массива не нужно было по всему коду заменять число 5 на новый размер, можно при помощи директивы `#define` определить символ, который будет заменяться на нужное число всюду в коде программы. Делается это так:

```
#define N 5

/* далее в коде, например: */
int A[N];

for (i = 0; i < N; i++)
    ...
```

Обратите внимание: в строке `#define` **нет точки с запятой**; если её по ошибке поставить — она будет подставляться вместо `N` вместе с пятёркой и мы получим, например, неверное определение массива `int A[5;]`; такие ошибки бывает весьма сложно найти.

6 Задачи на работу с массивами

Задача. Перепишите функцию `reverse` без рекурсии при помощи массива символов. Считайте, что длина ввода ограничена некоторым числом, чтобы завести массив такого размера.

Задача. Найдите максимальный элемент в данном массиве.

Задача. Напечатайте первые N простых чисел, используя решето Эратосфена, т. е. вычёркивая сначала чётные числа, затем числа, кратные трём, кратные пяти и т. д. Используйте массив, изначально заполненный нулями.

7 Проверочное задание («пятиминутка»)

Тема: работа с одномерными массивами.

1. Дан массив `A`. Вычислите и напечатайте среднее арифметическое его элементов.
2. Дан массив `A`. Вычислите и напечатайте количество его положительных элементов.
3. Дан массив `A`. Вычислите и напечатайте количество его нечётных элементов.
4. Дан массив `A`. Вычислите и напечатайте количество его элементов, равных нулю.