
Обзор функций ввода-вывода

А. Г. Фенстер, `fenster@fenster.name`

6 марта 2010 г.

Цель этого занятия — не пересказать несколько мануалов, а обратить внимание на наиболее частые проблемы и ответить на частые вопросы, возникающие при работе с функциями ввода-вывода.

Информация о наиболее часто используемых функциях ввода-вывода: <http://info.fenster.name/c/files.pdf>.

1 Функции семейства `printf`

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...); /* C99 */
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap); /* C99 */
```

Все `printf`-подобные функции возвращают количество символов, которые были или были **бы** напечатаны (последнее касается функций, принимающих параметр `size` — это `snprintf` и `vsnprintf`; см. далее).

Если нужно напечатать строку из переменной `s`, нельзя передавать эту строку в первый аргумент функции `printf`, поскольку в ней могут встречаться символы `%`. Если эта строка содержит ввод пользователя, это создаёт проблему безопасности, поскольку `printf` поддерживает флаг преобразования `%n`, который сохраняет в очередной аргумент (типа `int *`) количество напечатанных к текущему моменту символов. Следующий код сохранит в переменной `a` число 5, а в `b` — 10:

```
int a, b;
/* _set_printf_count_output(1); - требуется для Microsoft Visual Studio */
printf("vasya\npetya\n", &a, &b);
```

Ввиду (якобы) небезопасности `%n` этот флаг отключен в Microsoft Visual Studio по умолчанию (использование вызывает ошибку `Assertion failed`) и включается при помощи `_set_printf_count_output`, как показано выше. Естественно, это поведение стандарту не соответствует.

Функции семейства `printf`, имеющие в своём имени букву `s`, не печатают текст в поток вывода, а сохраняют его в строке `char *`. К сожалению, зачастую нет возможности предсказать, сколько символов этот текст займёт, и выделить соответствующее количество памяти под строку; зачастую использование `sprintf` может привести к переполнению буфера (нужно быть аккуратным). В C99 появились функции `snprintf` и `vsnprintf`, принимающие параметром число `size` и не сохраняющие в строку больше, чем `size` символов, но Visual Studio не поддерживает C99 и не предоставляет `snprintf` (зато предоставляет свою собственную `sprintf_s`). Если даже размера строки не хватило, функции `snprintf` и `vsnprintf` вернут количество символов, необходимых для печати (для того, чтобы можно было увеличить буфер).

Функции, принимающие `va_list`, мы кратко упоминали на прошлом занятии.

2 Функции семейства `scanf`

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

Все `scanf`-подобные функции возвращают количество **значений** (не символов), которые были прочитаны и сохранены. В случае наступления конца файла или несовпадения ввода с требуемым форматом функция может вернуть как `EOF` (если не успела считать и сохранить ни одного значения), так и неотрицательное число — количество значений, которые были успешно сохранены до этого момента. Соответственно, наиболее удобным способом проверить, что `scanf` отработал корректно, является сравнение его результата с количеством требуемых переменных:

```
if (scanf("%d%d", &a, &b) != 2)
    ошибка чтения;
```

Использование `%s` может очевидным образом привести к проблеме безопасности (невозможно предсказать, сколько символов будет введе-

но и хватит ли данной строки для того, чтобы их сохранить). Соответственно, необходимо использовать явное указание максимальной длины, например, `%20s`.

Как и многие другие функции, `scanf`-подобные функции объявлены устаревшими в Microsoft Visual Studio, начиная с MSVS 2005. Вместо него предлагается использовать `scanf_s`, требующий указывать размеры строковых полей в явном виде. Лично мне не кажется, что подобные ограничения могут спасти плохой код и это очень просто объясняется: человек, который заботится о том, чтобы выделенной памяти хватило, думает об этом и в случае использования стандартных («устаревших») функций, тогда как для неаккуратного программиста ни лишний warning, ни необходимость указать размер буфера помехой к написанию плохого (опасного) кода не являются.

3 Функции для ввода текста

```
int getchar(void); /* возвр. символ или EOF */
int fgetc(FILE *stream); /* возвр. символ или EOF */
char *fgets(char *s, int size, FILE *stream);
int ungetc(int c, FILE *stream);
```

Функция `gets` намеренно не упоминается: использовать её нельзя.

Обратите внимание, что все функции, вводящие символ, возвращают `int`, а не `char`, поскольку помимо любого из 256 символов они могут вернуть «ошибку» — константу `EOF` (обычно равную `-1` в типе `int`).

Проблемы возникают, если выполняются одновременно два условия:

1. Программист по ошибке использует тип `char` для чтения символов;
2. На ввод программе подаются русские буквы или любые другие символы с кодом `> 128`.

Подробнее об этих проблемах чуть ниже.

Функция `fgets` отличается тем, что сохраняет в строку символ перевода строки `'\n'`, который в подавляющем большинстве случаев там совершенно не нужен, но другого способа определить, закончилась ли строка или нужно читать дальше, поскольку не хватило буфера, нет. Функция возвращает тот же адрес буфера, что был передан ей параметром, или `NULL` в случае конца файла, что даёт возможность использовать её в цикле

```
while (fgets(buf, BUFSIZE, f))
{
    ...
}
```

Функция `ungetc` может вернуть **один** символ обратно в поток. Возможность вернуть даже один символ оказывает неоценимую услугу при разработке лексических анализаторов (например, арифметическое выражение можно читать посимвольно; число читать в цикле «пока прочитанный символ — цифра», а если прочитана не цифра — смело вернуть её обратно в поток, не разбирая, что там за символ: пробел, знак операции скобка или что-то ещё — этот разбор будет сделан при следующем чтении символа в более подходящем для этого месте кода).

3.1 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’ (поряд)
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 рассматривать 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', ... По историческим причинам именно KOI8-R распространена в UNIX-подобных операционных системах.
- Кодировка CP1251, она же Windows-1251, используется в одно-

имённой 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.

4 Функции для вывода текста

```
int fputc(int c, FILE *stream); /* возвр. символ или EOF */
int fputs(const char *s, FILE *stream); /* возвр. >=0 или EOF */
int putchar(int c); /* возвр. символ или EOF */
int puts(const char *s); /* возвр. >=0 или EOF */
```

Небольшой нюанс: `puts` переводит строку (печатает `'\n'` в конце), `fputs` этого не делает.

5 Функции для бинарного ввода-вывода

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Обратите внимание, что порядок параметров `size` и `nmemb` **не таковой**, как в функции `qsort`. Можно попытаться запомнить это так: для сортировки важнее количество элементов в массиве, для чтения важнее размер одного элемента.

Функции возвращают количество прочитанных или записанных записей размера `size`. При использовании `fread` в цикле нужно повторять чтение, пока функция возвращает ровно такое количество записей, сколько было запрошено. Если результат функции равен 0 или меньше запрошенного количества — файл закончился или произошла ошибка.