Глава 7. Ввод и вывод

7.1 Стандартный ввод-вывод
7.2 Форматный вывод (**printf**)
7.3 Списки аргументов переменной длины
7.4 Форматный ввод (**scanf**)
7.5 Доступ к файлам
7.6 Управление ошибками (**stderr** и **exit**)
7.7 Ввод-вывод строк
7.8 Другие библиотечные функции
7.8.1 Операции со строками
7.8.2 Анализ класса символов и преобразование символов
7.8.3 Функция **ungetc**
7.8.4 Исполнение команд операционной системы
7.8.5 Управление памятью
7.8.6 Математические функции
7.8.7 Генератор случайных чисел

Возможности для ввода и вывода не являются частью самого языка Си, поэтому мы подробно и не рассматривали их до сих пор. Между тем реальные программы взаимодействуют со своим окружением гораздо более сложным способом, чем те, которые были затронуты ранее. В этой главе мы опишем стандартную библиотеку, содержащую набор функций, обеспечивающих ввод-вывод, работу со строками, управление памятью, стандартные математические функции и разного рода сервисные Си-программы. Но особое внимание уделим вводу-выводу.

Библиотечные функции ввода-вывода точно определяются стандартом ANSI, так что они совместимы на любых системах, где поддерживается Си. Программы, которые в своем взаимодействии с системным окружением не выходят за рамки возможностей стандартной библиотеки, можно без изменений переносить с одной машины на другую.

Свойства библиотечных функций специфицированы в более чем дюжине заголовочных файлов; вам уже встречались некоторые из них, в том числе <**stdio.h**>, <**string.h**> и <**ctype.h**>. Мы не рассматриваем здесь всю библиотеку, так как нас больше интересует написание Си-программ, чем использование библиотечных функций. Стандартная библиотека подробно описана в приложении B.

7.1 Стандартный ввод-вывод

Как уже говорилось в главе 1, библиотечные функции реализуют простую модель текстового ввода-вывода. Текстовый поток состоит из последовательности строк; каждая строка заканчивается символом новой строки. Если система в чем-то не следует принятой модели, библиотека сделает так, чтобы казалось, что эта модель удовлетворяется полностью. Например, пара символов - *возврат-каретки* и *перевод-строки* - при вводе могла бы быть преобразована в один символ новой строки, а при выводе выполнялось бы обратное преобразование.

Простейший механизм ввода - это чтение одного символа из *стандартного ввода* (обычно с клавиатуры) функцией *getchar*:

```
int getchar(void)
```

В качестве результата каждого своего вызова функция *getchar* возвращает следующий символ ввода или, если обнаружен конец файла, **EOF**. Именованная константа EOF (аббревиатура от *end of file* - конец файла) определена в <**stdio.h**>. Обычно значение **EOF** равно -1, но, чтобы не зависеть от конкретного значения этой константы, обращаться к ней следует по имени (EOF).

Во многих системах клавиатуру можно заменить файлом, перенаправив ввод с помощью значка <. Так, если программа *prog* использует *getchar*, то командная строка

```
prog < infile
```

предпишет программе *prog* читать символы из *infile*, а не с клавиатуры. Переключение ввода делается так, что сама программа *prog* не замечает подмены; в частности строка *"< infile"* не будет включена в аргументы командной строки *argv*. Переключение ввода будет также незаметным, если ввод исходит от другой программы и передается конвейерным образом. В некоторых системах командная строка

```
otherprog | prog
```

приведет к тому, что запустится две программы, *otherprog* и *prog*, и стандартный выход *otherprog* поступит на стандартный вход *prog*. Функция

```
int putchar(int)
```

используется для вывода: *putchar(c)* отправляет символ *c* в *стандартный вывод*, под которым по умолчанию подразумевается экран. Функция *putchar* в качестве результата возвращает посланный символ или, в случае ошибки, *EOF*. То же и в отношении вывода: с помощью записи вида > *имя-файла* вывод можно перенаправить в файл. Например, если *prog* использует для вывода функцию *putchar*, то

```
prog > outfile
```

будет направлять стандартный вывод не на экран, а в *outfile*. А командная строка

```
prog | anotherprog
```

соединит стандартный вывод программы *prog* со стандартным вводом программы *anotherprog*.

Вывод, осуществляемый функцией *printf*, также отправляется в стандартный выходной поток. Вызовы *putchar* и *printf* могут как угодно чередоваться, при этом вывод будет формироваться в той последовательности, в которой происходили вызовы этих функций.

Любой исходный Си-файл, использующий хотя бы одну функцию библиотеки ввода-вывода, должен содержать в себе строку

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

причем она должна быть расположена до первого обращения к вводу-выводу. Если имя заголовочного файла заключено в угловые скобки < и >, это значит, что поиск заголовочного файла ведется в стандартном месте (например в системе UNIX это обычно директорий */usr/include*).

Многие программы читают только из одного входного потока и пишут только в один выходной поток. Для организации ввода-вывода таким программам вполне хватит функций *getchar*, *putchar* и *printf*, а для начального обучения уж точно достаточно ознакомления с этими функциями. В частности, перечисленных функций достаточно, когда требуется вывод одной программы соединить с вводом следующей. В качестве примера рассмотрим программу *lower*, переводящую свой ввод на нижний регистр:

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

main() /* lower: переводит ввод на нижний регистр */
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Функция *tolower* определена в <**ctype.h**>. Она переводит буквы верхнего регистра в буквы нижнего

регистра, а остальные символы возвращает без изменений. Как мы уже упоминали, "функции" вроде *getchar* и *putchar* из библиотеки **<stdio.h>** и функция *tolower* из библиотеки **<ctype.h>** часто реализуются в виде макросов, чтобы исключить накладные расходы от вызова функции на каждый отдельный символ. В параграфе 8.5 мы покажем, как это делается. Независимо от того, как на той или иной машине реализованы функции библиотеки **<ctype.h>**, использующие их программы могут ничего не знать о кодировке символов.

Упражнение 7.1. Напишите программу, осуществляющую перевод ввода с верхнего регистра на нижний или с нижнего на верхний в зависимости от имени, по которому она вызывается и текст которого находится в *arg[0]*.

7.2 Форматный вывод (printf)

Функция **printf** переводит внутренние значения в текст.

```
int printf(char *format, arg1, arg2, ...)
```

В предыдущих главах мы использовали *printf* неформально. Здесь мы покажем наиболее типичные случаи применения этой функции: полное ее описание дано в приложении B.

Функция *printf* преобразует, форматирует и печатает свои аргументы в стандартном выводе под управлением формата. Возвращает она количество напечатанных символов.

Форматная строка содержит два вида объектов: обычные символы, которые напрямую копируются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента *printf*. Любая спецификация преобразования начинается знаком **%** и заканчивается *символом-спецификатором*. Между % и символом-спецификатором могут быть расположены (в указанном ниже порядке) следующие элементы:

- Знак минус, предписывающий выравнивать преобразованный аргумент по левому краю поля.
- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет занимать поле по крайней мере указанной ширины. При необходимости лишние позиции слева (или справа при левостороннем расположении) будут заполнены пробелами.
- Точка, отделяющая ширину поля от величины, устанавливающей точность.
- Число (точность), специфицирующее максимальное количество печатаемых символов в строке, или количество цифр после десятичной точки - для чисел с плавающей запятой, или минимальное количество цифр - для целого.
- Буква *h*, если печатаемое целое должно рассматриваться как *short*, или *l* (латинская буква ell), если целое должно рассматриваться как *long*.

Символы-спецификаторы перечислены в таблице 7.1. Если за **%** не помещен символ- спецификатор, поведение функции *printf* будет не определено. Ширину и точность можно специфицировать с помощью *****; значение ширины (или точности) в этом случае берется из следующего аргумента (который должен быть типа *int*). Например, чтобы напечатать не более *max* символов из строки *s*, годится следующая запись:

```
printf("%.*s", max, s);
```

Таблица 7.1 Основные преобразования printf

Символ	Тип аргумента; вид печати
d, **i**	**int**; десятичное целое
o	**unsigned int**; беззнаковое восьмеричное (*octal*) целое (без нуля слева)
x, **X**	**unsigned int**; беззнаковое шестнадцатеричное целое (без 0x или 0X слева), для 10...15 используются abcdef или ABCDEF
u	**unsigned int**; беззнаковое десятичное целое
c	**int**; одиночный символ
s	**char ***; печатает символы, расположенные до знака \0, или в количестве, заданном точностью

f	**double**; **[-]m.dddddd**, где количество цифр **d** задается точностью (по умолчанию равно *6*)
e, **E**	**double**; **[-]m.ddddddе±xx** или **[-]m.ddddddE±xx**, где количество цифр **d** задается точностью (по умолчанию равно *6*)
g, **G**	**double**; использует **%e** или **%E**, если порядок меньше, чем -4, или больше или равен точности; в противном случае использует **%f**. Завершающие нули и завершающая десятичная точка не печатаются
p	**void ***; указатель (представление зависит от реализации)
%	Аргумент не преобразуется; печатается знак **%**

Большая часть форматных преобразований была продемонстрирована в предыдущих главах. Исключение составляет задание точности для строк. Далее приводится перечень спецификаций и показывается их влияние на печать строки "hello, world", состоящей из 12 символов. Поле специально обрамлено двоеточиями, чтобы была видна его протяженность.

```
:%s:        :hello, world:
:%10s       :hello, world:
:%.10s:     :hello, wor:
:%-10s:     :hello, world:
:%.15s:     :hello, world:
:%-15s:     :hello, world     :
:%15.10s:   :       hello, wor:
:%-15.10s:  :hello, wor       :
```

Предостережение: функция *printf* использует свой первый аргумент, чтобы определить, сколько еще ожидается аргументов и какого они будут типа. Вы не получите правильного результата, если аргументов будет не хватать или они будут принадлежать не тому типу. Вы должны также понимать разницу в следующих двух обращениях:

```
printf(s); /* НЕВЕРНО, если в s есть % */
printf("%s", s); /* ВЕРНО всегда */
```

Функция **sprintf** выполняет те же преобразования, что и *printf*, но вывод запоминает в строке

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

Эта функция форматирует arg_1, arg_2 и т. д. в соответствии с информацией, заданной аргументом *format*, как мы описывали ранее, но результат помещает не в стандартный вывод, а в *string*. Заметим, что строка *string* должна быть достаточно большой, чтобы в ней поместился результат.

Упражнение 7.2. Напишите программу, которая будет печатать разумным способом любой ввод. Как минимум она должна уметь печатать неграфические символы в восьмеричном или шестнадцатеричном виде (в форме, принятой на вашей машине), обрывая длинные текстовые строки.

7.3 Списки аргументов переменной длины

Этот параграф содержит реализацию минимальной версии *printf*. Приводится она для того, чтобы показать, как надо писать функции со списками аргументов переменной длины, причем такие, которые были бы переносимы. Поскольку нас главным образом интересует обработка аргументов, функцию *minprintf* напишем таким образом, что она в основном будет работать с задающей формат строкой и аргументами; что же касается форматных преобразований, то они будут осуществляться с помощью стандартного *printf*.

Объявление стандартной функции *printf* выглядит так:

```
int printf(char *fmt, ...)
```

Многоточие в объявлении означает, что число и типы аргументов могут изменяться. Знак многоточие может стоять только в конце списка аргументов. Наша функция *minprintf* объявляется как

```
void minprintf(char *fmt, ...)
```

поскольку она не будет выдавать число символов, как это делает *printf*.

Вся сложность в том, каким образом *minprintf* будет продвигаться вдоль списка аргументов, - ведь у этого списка нет даже имени. Стандартный заголовочный файл <**stdarg.h**> содержит набор макроопределений, которые устанавливают, как шагать по списку аргументов. Наполнение этого заголовочного файла может изменяться от машины к машине, но представленный им интерфейс везде одинаков.

Тип **va_list** служит для описания переменной, которая будет по очереди указывать на каждый из аргументов; в *minprintf* эта переменная имеет имя *ap* (от "*argument pointer*" - указатель на аргумент). Макрос **va_start** инициализирует переменную *ap*) чтобы она указывала на первый безымянный аргумент. К **va_start** нужно обратиться до первого использования *ap*. Среди аргументов по крайней мере один должен быть именованным: от последнего именованного аргумента этот макрос "отталкивается" при начальной установке.

Макрос **va_arg** на каждом своем вызове выдает очередной аргумент, а *ap* передвигает на следующий: но имени типа он определяет тип возвращаемого значения и размер шага для выхода на следующий аргумент. Наконец, макрос **va_end** делает очистку всего, что необходимо. К **va_end** следует обратиться перед самым выходом из функции.

Перечисленные средства образуют основу нашей упрощенной версии *prinf*.

```
#include <stdarg.h>

/* minprintf: минимальный printf с переменным числом аргументов */
void minprintf(char *fmt, ...)
{
    va_list ap;        /* указывает на очередной безымянный аргумент */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* устанавливает ap на 1-й безымянный аргумент */
    for (p=fmt; *p; p++) {
        if (*p !='%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf ("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap); /* очистка, когда все сделано */
}
```

Упражнение 7.3. Дополните *minprintf* другими возможностями *printf*.

7.4 Форматный ввод (scanf)

Функция **scanf**, обеспечивающая ввод, является аналогом *printf*; она выполняет многие из упоминавшихся преобразований, но в противоположном направлении. Ее объявление имеет следующий вид:

```
int scanf(char *format, ...)
```

Функция *scanf* читает символы из стандартного входного потока, интерпретирует их согласно спецификациям строки *format* и рассылает результаты в свои остальные аргументы. Аргумент *format* мы опишем позже; другие аргументы, каждый из которых должен быть указателем, определяют, где будут запоминаться должным образом преобразованные данные. Как и для *printf*, в этом параграфе

дается сводка наиболее полезных, но отнюдь не всех возможностей данной функции.

Функция *scanf* прекращает работу, когда оказывается, что исчерпался формат или вводимая величина не соответствует управляющей спецификации. В качестве результата *scanf* возвращает количество успешно введенных элементов данных. По исчерпании файла она выдает *EOF*. Существенно то, что значение EOF не равно нулю, поскольку нуль *scanf* выдает, когда вводимый символ не соответствует первой спецификации форматной строки. Каждое очередное обращение к *scanf* продолжает ввод символа, следующего сразу за последним обработанным.

Существует также функция *sscanf*, которая читает из строки (а не из стандартного ввода).

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Функция *sscanf* просматривает строку *string* согласно формату *format* и рассылает полученные значения в arg_1, arg_2 и т. д. Последние должны быть указателями.

Формат обычно содержит спецификации, которые используются для управления преобразованиями ввода. В него могут входить следующие элементы:

- Пробелы или табуляции, которые игнорируются.
- Обычные символы (исключая %), которые, как ожидается, совпадут с очередными символами, отличными от символов-разделителей входного потока.
- Спецификации преобразования, каждая из которых начинается со знака % и завершается символом-спецификатором типа преобразования. В промежутке между этими двумя символами в любой спецификации могут располагаться, причем в том порядке, как они здесь указаны: знак * (признак подавления присваивания); число, определяющее ширину поля; буква *h*, *l* или *L*, указывающая на размер получаемого значения; и символ преобразования (*o*, *d*, *x*).

Спецификация преобразования управляет преобразованием следующего вводимого поля. Обычно результат помещается в переменную, на которую указывает соответствующий аргумент. Однако если в спецификации преобразования присутствует *, то поле ввода пропускается и никакое присваивание не выполняется. Поле ввода определяется как строка без символов-разделителей; оно простирается до следующего символа-разделителя или же ограничено шириной поля, если она задана. Поскольку символ новой строки относится к символам- разделителям, то *sscanf* при чтении будет переходить с одной строки на другую. (**Символами-разделителями являются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и перевода страницы.**)

Символ-спецификатор указывает, каким образом следует интерпретировать очередное поле ввода. Соответствующий аргумент должен быть указателем, как того требует механизм передачи параметров по значению, принятый в Си. Символы-спецификаторы приведены в таблице 7.2.

Перед символами-спецификаторами **d**, **l**, **o**, **u** и **x** может стоять буква **h**, указывающая на то, что соответствующий аргумент должен иметь тип *short* * (а не *int* *),или **l** (латинская ell), указывающая на тип *long* *. Аналогично, перед символами-спецификаторами **e**, **f** и **g** может стоять буква **l**, указывающая, что тип аргумента - *double* * (а не *float* *).

Таблица 7.2 Основные преобразования scanf

Символ	Вводимые данные; тип аргумента
d	десятичное целое: **int ***
i	целое: **int ***. Целое может быть восьмеричным (с 0 слева) или шестнадцатеричным (с 0x или 0X слева)
o	восьмеричное целое (с нулем слева или без него); **int ***
u	беззнаковое десятичное целое; **unsigned int ***
x	шестнадцатеричное целое (с 0x или 0X слева или без них); **int ***
c	символы; **char ***. Следующие символы ввода (по умолчанию один) размещаются в указанном месте. Обычный пропуск символов- разделителей подавляется; чтобы прочесть очередной символ, отличный от символа-разделителя, используйте %1s

s	Строка символов(без обрамляющих кавычек); **char ***, указывающая на массив символов, достаточный для строки и завершающего символа '\0', который будет добавлен
e, **f**, **g**	число с плавающей точкой, возможно, со знаком; обязательно присутствие либо десятичной точки, либо экспоненциальной части, а возможно, и обеих вместе; **float ***
%	сам знак **%**, никакое присваивание не выполняется

Чтобы построить первый пример, обратимся к программе калькулятора из главы 4, в которой организуем ввод с помощью функции *scanf*:

```
#include <stdio.h>
main() /* программа-калькулятор */
{
    double sum, v;

    sum = 0;
    while (scanf ("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Предположим, что нам нужно прочитать строки ввода, содержащие данные вида

```
25 дек 1988
```

Обращение к *scanf* выглядит следующим образом:

```
int day, year; /* день, год */
char monthname[20]; /* название месяца */

scanf ("%d %s %d", &day, monthname, &year);
```

Знак *&* перед *monthname* не нужен, так как имя массива есть указатель.

В строке формата могут присутствовать символы, не участвующие ни в одной из спецификаций; это значит, что эти символы должны появиться на вводе. Так, мы могли бы читать даты вида *mm/dd/yy* с помощью следующего обращения к *scanf*:

```
int day, month, year; /* день, месяц, год */
scanf("%d/%d/%d", &day, &month, &year);
```

В своем формате функция *scanf* игнорирует пробелы и табуляции. Кроме того, при поиске следующей порции ввода она пропускает во входном потоке все символы- разделители (пробелы, табуляции, новые строки и т.д.). Воспринимать входной поток, не имеющий фиксированного формата, часто оказывается удобнее, если вводить всю строку целиком и для каждого отдельного случая подбирать подходящий вариант *sscanf*. Предположим, например, что нам нужно читать строки с датами, записанными в любой из приведенных выше форм. Тогда мы могли бы написать:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("верно: %s\r", line); /* в виде 25 дек 1968 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("верно: %s\n", line); /* в виде mm/dd/yy */
    else
        printf("неверно: %s\n", line); /* неверная форма даты */
}
```

Обращения к *scanf* могут перемежаться с вызовами других функций ввода. Любая функция ввода, вызванная после *scanf*, продолжит чтение с первого еще непрочитанного символа.

В завершение еще раз напомним, что аргументы функций *scanf* и *sscanf* должны быть указателями.

Одна из самых распространенных ошибок состоит в том, что вместо того, чтобы написать

```
scanf ("%d", &n);
```

пишут

```
scanf("%d", n);
```

Компилятор о подобной ошибке ничего не сообщает.

Упражнение 7.4. Напишите свою версию *scanf* по аналогии с *minprintf* из предыдущего параграфа.

Упражнение 7.5. Перепишите основанную на постфиксной записи программу калькулятора из главы 4 таким образом, чтобы для ввода и преобразования чисел она использовала *scanf* и/или *sscanf*.

7.5 Доступ к файлам

Во всех предыдущих примерах мы имели дело со стандартным вводом и стандартным выводом, которые для программы автоматически предопределены операционной системой конкретной машины.

Следующий шаг - научиться писать программы, которые имели бы доступ к файлам, заранее не подсоединенным к программам. Одна из программ, в которой возникает такая необходимость, - это программа *cat*, объединяющая несколько именованных файлов и направляющая результат в стандартный вывод. Функция *cat* часто применяется для выдачи файлов на экран, а также как универсальный "коллектор" файловой информации для тех программ, которые не имеют возможности обратиться к файлу по имени. Например, команда

```
cat x.c y.c
```

направит в стандартный вывод содержимое файлов *x.c* и *y.c* (и ничего более).

Возникает вопрос: что надо сделать, чтобы именованные файлы можно было читать; иначе говоря, как связать внешние имена, придуманные пользователем, с инструкциями чтения данных?

На этот счет имеются простые правила. Для того чтобы можно было читать из файла или писать в файл, он должен быть предварительно открыт с помощью библиотечной функции **fopen**. Функция *fopen* получает внешнее имя типа *x.c* или *y.c*, после чего осуществляет некоторые организационные действия и "переговоры" с операционной системой (технические детали которых здесь не рассматриваются) и возвращает указатель, используемый в дальнейшем для доступа к файлу.

Этот указатель, называемый *указателем файла*, ссылается на структуру, содержащую информацию о файле (адрес буфера, положение текущего символа в буфере, открыт файл на чтение или на запись, были ли ошибки при работе с файлом и не встретился ли конец файла). Пользователю не нужно знать подробности, поскольку определения, полученные из <stdio.h>, включают описание такой структуры, называемой **FILE**.

Единственное, что требуется для определения указателя файла, - это задать описания такого, например, вида:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Это говорит, что *fp* есть указатель на *FILE*, а *fopen* возвращает указатель на *FILE*. Заметим, что *FILE* — это имя типа) наподобие *int*, а не тег структуры. Оно определено с помощью **typedef**. (Детали того, как можно реализовать *fopen* в системе UNIX, приводятся в параграфе 8.5.)

Обращение к *fopen* в программе может выглядеть следующим образом:

```
fp = fopen(name, mode);
```

Первый аргумент - строка, содержащая имя файла. Второй аргумент несет информацию о режиме. Это тоже строка: в ней указывается, каким образом пользователь намерен применять файл. Возможны следующие режимы: чтение (*read* - "r"), запись (*write* - "w") и добавление (*append* - "a"), т. е. запись информации в конец уже существующего файла. В некоторых системах различаются текстовые и бинарные файлы; в случае последних в строку режима необходимо добавить букву "b" (*binary* - бинарный).

Тот факт, что некий файл, которого раньше не было, открывается на запись или добавление, означает, что он создается (если такая процедура физически возможна). Открытие уже существующего файла на запись приводит к выбрасыванию его старого содержимого, в то время как при открытии файла на добавление его старое содержимое сохраняется. Попытка читать несуществующий файл является

ошибкой. Могут иметь место и другие ошибки; например, ошибкой считается попытка чтения файла, который по статусу запрещено читать. При наличии любой ошибки *fopen* возвращает NULL. (Возможна более точная идентификация ошибки; детальная информация по этому поводу приводится в конце параграфа 1 приложения B.)

Следующее, что нам необходимо знать, - это как читать из файла или писать в файл, коль скоро он открыт. Существует несколько способов сделать это, из которых самый простой состоит в том, чтобы воспользоваться функциями **getc** и **putc**. Функция *getc* возвращает следующий символ из файла; ей необходимо сообщить указатель файла, чтобы она знала откуда брать символ.

```
int getc(FILE *fp);
```

Функция *getc* возвращает следующий символ из потока, на который указывает **fp*; в случае исчерпания файла или ошибки она возвращает EOF.

Функция *putc* пишет символ *c* в файл *fp*

```
int putc(int c, FILE *fp);
```

и возвращает записанный символ или EOF в случае ошибки. Аналогично *getchar* и *putchar*, реализация *getc* и *putc* может быть выполнена в виде макросов, а не функций.

При запуске Си-программы операционная система всегда открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок; соответствующие им указатели называются **stdin**, **stdout** и **stderr**; они описаны в <stdio.h>. Обычно *stdin* соотнесен с клавиатурой, а *stdout* и *stderr* - с экраном. Однако *stdin* и *stdout* можно связать с файлами или, используя конвейерный механизм, соединить напрямую с другими программами, как это описывалось в параграфе 7.1.

С помощью *getc*, *putc*, *stdin* и *stdout* функции *getchar* и *putchar* теперь можно определить следующим образом:

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

Форматный ввод-вывод файлов можно построить на функциях **fscanf** и **fprintf**. Они идентичны *scanf* и *printf* с той лишь разницей, что первым их аргументом является указатель на файл, для которого осуществляется ввод-вывод, формат же указывается вторым аргументом.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Вот теперь мы располагаем теми сведениями, которые достаточны для написания программы *cat*, предназначенной для конкатенации (последовательного соединения) файлов. Предлагаемая версия функции *cat*, как оказалось, удобна для многих программ. Если в командной строке присутствуют аргументы, они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается стандартный ввод.

```
#include <stdio.h>
/* cat: конкатенация файлов, версия 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc == 1) /* нет аргументов; копируется стандартный ввод */
        filecopy(stdin, stdout);
    else
    while (--argc > 0)
        if ((fp = fopen(*++argv, "r")) == NULL) {
            printf("cat: не могу открыть файл %s\n", *argv);
            return 1;
        } else {
            filecopy(fp, stdout);
            fclose(fp);
        }
    return 0;
}

/* filecopy: копирует файл ifp в файл ofp */
```

```
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Файловые указатели *stdin* и *stdout* представляют собой объекты типа *FILE**. Это константы, а не переменные, следовательно, им нельзя ничего присваивать.

Функция

```
int fclose(FILE *fp)
```

- обратная по отношению к *fopen*; она разрывает связь между файловым указателем и внешним именем (которая раньше была установлена с помощью *fopen*), освобождая тем самым этот указатель для других файлов. Так как в большинстве операционных систем количество одновременно открытых одной программой файлов ограничено, то файловые указатели, если они больше не нужны, лучше освобождать, как это и делается в программе *cat*. Есть еще одна причина применить *fclose* к файлу вывода, - это необходимость "опорожнить" буфер, в котором *putc* накопила предназначенные для вывода данные. При нормальном завершении работы программы для каждого открытого файла *fclose* вызывается автоматически. (Вы можете закрыть *stdin* и *stdout*, если они вам не нужны. Воспользовавшись библиотечной функцией **freopen**, их можно восстановить.)

7.6 Управление ошибками (stderr и exit)

Обработку ошибок в *cat* нельзя признать идеальной. Беда в том, что если файл по какой-либо причине недоступен, сообщение об этом мы получим по окончании конкатенируемого вывода. Это нас устроило бы, если бы вывод отправлялся только на экран, а не в файл или по конвейеру другой программе. Чтобы лучше справиться с этой проблемой, программе помимо стандартного вывода *stdout* придается еще один выходной поток, называемый *stderr*. Вывод в *stderr* обычно отправляется на экран, даже если вывод *stdout* перенаправлен в другое место. Перепишем *cat* так, чтобы сообщения об ошибках отправлялись в *stderr*.

```
#include <stdio.h>
/* cat: конкатенация файлов, версия 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* имя программы */
    if (argc == 1)        /* нет аргументов, копируется станд. ввод */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: не могу открыть файл %s\n", prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: ошибка записи в stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

Программа сигнализирует об ошибках двумя способами. Первый - сообщение об ошибке с помощью *fprintf* посылается в *stderr* с тем, чтобы оно попало на экран, а не оказалось на конвейере или в другом файле вывода. Имя программы, хранящееся в *argv[0]*, мы включили в сообщение, чтобы в случаях, когда данная программа работает совместно с другими, был ясен источник ошибки.

Второй способ указать на ошибку - обратиться к библиотечной функции **exit**, завершающей работу программы. Аргумент функции *exit* доступен некоторому процессу, вызвавшему данный процесс. А следовательно, успешное или ошибочное завершение программы можно проконтролировать с помощью некоей программы, которая рассматривает эту программу в качестве подчиненного процесса. По общей договоренности возврат нуля сигнализирует о том, что работа прошла нормально, в то время

как ненулевые значения обычно говорят об ошибках. Чтобы опорожнить буфера, накопившие информацию для всех открытых файлов вывода, функция *exit* вызывает *fclose*.

Инструкция **return** *exp* главной программы *main* эквивалентна обращению к функции *exit(exp)*. Последний вариант (с помощью *exit*) имеет то преимущество, что он пригоден для выхода и из других функций, и, кроме того, слово *exit* легко обнаружить с помощью программы контекстного поиска, похожей на ту, которую мы рассматривали в главе 5. Функция **ferror** выдает ненулевое значение, если в файле *fp* была обнаружена ошибка.

```
int ferror(FILE *fp)
```

Хотя при выводе редко возникают ошибки, все же они встречаются (например, оказался переполненным диск); поэтому в программах широкого пользования они должны тщательно контролироваться.

Функция **feof**(*FILE *fp*) аналогична функции *ferror*; она возвращает ненулевое значение, если встретился конец указанного в аргументе файла.

```
int feof(FILE *fp);
```

В наших небольших иллюстративных программах мы не заботились о выдаче статуса выхода, т. е. выдаче некоторого числа, характеризующего состояние программы в момент завершения: работа закончилась нормально или прервана из-за ошибки? Если работа прервана в результате ошибки, то какой? Любая серьезная программа должна выдавать статус выхода.

7.7 Ввод-вывод строк

В стандартной библиотеке имеется программа ввода **fgets**, аналогичная программе *getline*, которой мы пользовались в предыдущих главах.

```
char *fgets(char *line, int maxline, FILE *fp)
```

Функция *fgets* читает следующую строку ввода (включая и символ новой строки) из файла *fp* в массив символов *line*, причем она может прочитать не более *MAXLINE-1* символов. Переписанная строка дополняется символом '\0'. Обычно *fgets* возвращает *line*, а по исчерпании файла или в случае ошибки - NULL. (Наша *getline* возвращала длину строки, которой мы потом пользовались, и нуль в случае конца файла.)

Функция вывода **fputs** пишет строку (которая может и не заканчиваться символом новой строки) в файл.

```
int fputs(char *line, FILE *fp)
```

Эта функция возвращает EOF, если возникла ошибка, и неотрицательное значение в противном случае.

Библиотечные функции **gets** и **puts** подобны функциям *fgets* и *fputs*. Отличаются они тем, что оперируют только стандартными файлами *stdin* и *stdout*, и кроме того, *gets* выбрасывает последний символ '\n', а *puts* его добавляет.

Чтобы показать, что ничего особенного в функциях вроде *fgets* и *fputs* нет, мы приводим их здесь в том виде, в каком они существуют в стандартной библиотеке на нашей системе.

```
/* fgets: получает не более n символов из iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs= '\0';
    return (c == EOF && cs == s) ? NULL : s;
}
```

```
/* fputs: посылает строку s в файл iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

Стандарт определяет, что функция *ferror* возвращает в случае ошибки ненулевое значение; *fputs* в случае ошибки возвращает EOF, в противном случае - неотрицательное значение.

С помощью *fgets* легко реализовать нашу функцию *getline*:

```
/* getline: читает строку, возвращает ее длину */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```

Упражнение 7.6. Напишите программу, сравнивающую два файла и печатающую первую строку, в которой они различаются.

Упражнение 7.7. Модифицируйте программу поиска по образцу из главы 5 таким образом, чтобы она брала текст из множества именованных файлов, а если имен файлов в аргументах нет, то из стандартного ввода. Будет ли печататься имя файла, в котором найдена подходящая строка?

Упражнение 7.8. Напишите программу, печатающую несколько файлов. Каждый файл должен начинаться с новой страницы, предваряться заголовком и иметь свою нумерацию страниц.

7.8 Другие библиотечные функции

В стандартной библиотеке представлен широкий спектр различных функций. Настоящий параграф содержит краткий обзор наиболее полезных из них. Более подробно эти и другие функции описаны в приложении B.

7.8.1 Операции со строками

Мы уже упоминали функции *strlen*, *strcpy*, *strcat* и *strcmp*, описание которых даны в <string.h>. Далее, до конца пункта, предполагается, что *s* и *t* имеют тип *char **, *c* и *n* - тип *int*.

strcat(s,t)	- приписывает *t* в конец *s*.
strncat(s,t,n)	- приписывает *n* символов из *t* в конец *s*.
strcmp(s,t)	- возвращает отрицательное число, нуль или положительное число для *s* < *t*, *s* == *t* или *s* > *t*, соответственно.
strncmp(s,t,n)	- делает то же, что и *strcmp*, но количество сравниваемых символов не может превышать *n*
strcpy(s,t)	- копирует *t* в *s*.
strncpy(s,t,n)	- копирует не более *n* символов из *t* в *s*.
strlen(s)	- возвращает длину *s*.
strchr(s,c)	- возвращает указатель на первое появление символа *c* в *s* или, если *c* нет в *s*, NULL.
strrchr(s,c)	- возвращает указатель на последнее появление символа *c* в *s* или, если *c* нет в *s*, NULL.

7.8.2 Анализ класса символов и преобразование символов

Несколько функций из библиотеки <ctype.h> выполняют проверки и преобразование символов. Далее, до конца пункта, переменная *c* - это переменная типа *int*, которая может быть представлена значением *unsigned*, *char* или *EOF*. Все эти функции возвращают значения типа *int*.

isalpha(c) - не нуль, если *c* - буква; 0 в противном случае.
isupper(c) - не нуль, если *c* - буква верхнего регистра; 0 в противном случае.
islower(c) - не нуль, если *c* - буква нижнего регистра; 0 в противном случае.
isdigit(c) - не нуль, если *c* - цифра; 0 в противном случае.
isalnum(c) - не нуль, если или *isalpha(c)*, или *isdigit(c)* истинны; 0 в противном случае.
isspace(c) - не нуль, если *c* - символ пробела, табуляции, новой строки, возврата каретки, перевода страницы, вертикальной табуляции.
toupper(c) - возвращает *c*, приведенную к верхнему регистру.
tolower(c) - возвращает *c*, приведенную к нижнему регистру.

7.8.3 Функция ungetc

В стандартной библиотеке содержится более ограниченная версия функции *ungetch* по сравнению с той, которую мы написали в главе 4. Называется она *ungetc*. Эта функция, имеющая прототип

```
int ungetc(int c, FILE *fp)
```

отправляет символ *c* назад в файл *fp* и возвращает *c*, а в случае ошибки EOF. Для каждого файла гарантирован возврат не более одного символа. Функцию *ungetc* можно использовать совместно с любой из функций ввода вроде *scanf*, *getc*, *getchar* и т. д.

7.8.4 Исполнение команд операционной системы

Функция **system**(char *s) выполняет команду системы, содержащуюся в строке *s*, и затем возвращается к выполнению текущей программы.

Содержимое *s*, строго говоря, зависит от конкретной операционной системы. Рассмотрим простой пример: в системе UNIX инструкция

```
system("date");
```

вызовет программу *date*, которая направит дату и время в стандартный вывод. Функция возвращает зависящий от системы статус выполненной команды. В системе UNIX возвращаемый статус - это значение, переданное функцией *exit*.

7.8.5 Управление памятью

Функции **malloc** и **calloc** динамически запрашивают блоки свободной памяти. Функция *malloc*

```
void *malloc(size_t n)
```

возвращает указатель на *n* байт неинициализированной памяти или NULL, если запрос удовлетворить нельзя. Функция *calloc*

```
void *calloc(size_t n, size_t size)
```

возвращает указатель на область, достаточную для хранения массива из *n* объектов указанного размера (*size*), или NULL, если запрос не удается удовлетворить. Выделенная память обнуляется.

Указатель, возвращаемый функциями *malloc* и *calloc*, будет выдан с учетом выравнивания, выполненного согласно указанному типу объекта. Тем не менее к нему должна быть применена операция приведения к соответствующему типу (Как уже отмечалось (см. примеч. в параграфе 6.5), замечания о приведении типов значений, возвращаемых функциями *malloc* или *calloc*, — неверно. — *Примеч. авт.*), как это сделано в следующем фрагменте программы:

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

Функция **free(p)** освобождает область памяти, на которую указывает *p*, - указатель, первоначально полученный с помощью *malloc* или *calloc*. Никаких ограничений на порядок, в котором будет освобождаться память, нет, но считается ужасной ошибкой освобождение тех областей, которые не

были получены с помощью *calloc* или *malloc*.

Нельзя также использовать те области памяти, которые уже освобождены. Следующий пример демонстрирует типичную ошибку в цикле, освобождающем элементы списка.

```
for (p = head; p != NULL; p = p->next) /* НЕВЕРНО */
    free(p);
```

Правильным будет, если вы до освобождения сохраните то, что вам потребуется, как в следующем цикле:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

В параграфе 8.7 мы рассмотрим реализацию программы управления памятью вроде *malloc*, позволяющую освобождать выделенные блоки памяти в любой последовательности.

7.8.6 Математические функции

В **<math.h>** описано более двадцати математических функций. Здесь же приведены наиболее употребительные. Каждая из них имеет один или два аргумента типа *double* и возвращает результат также типа *double*.

sin(x)	- синус x, x в радианах
cos(x)	- косинус x, x в радианах
atan2(y,x)	- арктангенс y/x, y и x в радианах
exp(x)	- экспоненциальная функция e в степени x
log(x)	- натуральный (по основанию e) логарифм x (x>0)
log10(x)	- обычный (по основанию 10) логарифм x (x>0)
pow(x,y)	- x в степени y
sqrt(x)	- корень квадратный x (x > 0)
fabs(x)	- абсолютное значение x

7.8.7 Генератор случайных чисел

Функция **rand()** вычисляет последовательность псевдослучайных целых в диапазоне от нуля до значения, заданного именованной константой RAND_MAX, которая определена в <stdlib.h>. Привести случайные числа к значениям с плавающей точкой, большим или равным 0 и меньшим 1, можно по формуле

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Если в вашей библиотеке уже есть функция для получения случайных чисел с плавающей точкой, вполне возможно, что ее статистические характеристики лучше указанной.)

Функция **srand**(*unsigned*) устанавливает семя для *rand*. Реализации *rand* и *srand*, предлагаемые стандартом и, следовательно, переносимые на различные машины, рассмотрены в параграфе 2.7.

Упражнение 7.9. Реализуя функции вроде *isupper*, можно экономить либо память, либо время. Напишите оба варианта функции.

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]