Back to Chapter 6 -- Index -- Chapter 8

Chapter 7 - Input and Output

Input and output are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including `<stdio.h>`, `<string.h>`, and `<ctype.h>`. We will not present the entire library here, since we are more interested in writing C programs that use it. The library is described in detail in Appendix B.

7.1 Standard Input and Output

As we said in Chapter 1, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`:

```
int getchar(void)
```

`getchar` returns the next input character each time it is called, or `EOF` when it encounters end of file. The symbolic constant `EOF` is defined in `<stdio.h>`. The value is typically -1, bus tests should be written in terms of `EOF` so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the < convention for input redirection: if a program `prog` uses `getchar`, then the command line

```
prog <infile
```

causes `prog` to read characters from `infile` instead. The switching of the input is done in such a way that `prog` itself is oblivious to the change; in particular, the string ``<infile'' is not included in the command-line arguments in `argv`. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.

The function

```
int putchar(int)
```

is used for output: `putchar(c)` puts the character `c` on the `standard output`, which is by default the screen. `putchar` returns the character written, or `EOF` is an error occurs. Again, output can usually be directed to a file with *>filename*: if `prog` uses `putchar`,

```
prog >outfile
```

will write the standard output to `outfile` instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of `prog` into the standard input of `anotherprog`.

Output produced by `printf` also finds its way to the standard output. Calls to `putchar` and `printf` may be interleaved - output happens in the order in which the calls are made.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory `/usr/include`).

Many programs read only one input stream and write only one output stream; for such programs, input and output with `getchar`, `putchar`, and `printf` may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program `lower`, which converts its input to lower case:

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

main() /* lower: convert input to lower case*/
{
    int c

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, ``functions'' like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character. We will show how this is done in Section 8.5. Regardless of how the `<ctype.h>` functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`.

7.2 Formatted Output - printf

The output function `printf` translates internal values to characters. We have used `printf` informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, see Appendix B.

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

`printf` converts, formats, and prints its arguments on the standard output under control of the `format`. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`. Each conversion specification begins with a % and ends with a

conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.
- An `h` if the integer is to be printed as a `short`, or `l` (letter ell) if as a `long`.

Conversion characters are shown in Table 7.1. If the character after the % is not a conversion specification, the behavior is undefined.

***Table 7.1** Basic Printf Conversions*

Character	Argument type; Printed As
`d,i`	`int`; decimal number
`o`	`int`; unsigned octal number (without a leading zero)
`x,X`	`int`; unsigned hexadecimal number (without a leading `0x` or `0X`), using `abcdef` or `ABCDEF` for 10, ...,15.
`u`	`int`; unsigned decimal number
`c`	`int`; single character
`s`	`char *`; print characters from the string until a `'\0'` or the number of characters given by the precision.
`f`	`double`; `[-]`*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
`e,E`	`double`; `[-]`*m.dddddd*`e+/-`*xx* or `[-]`*m.dddddd*`E+/-`*xx*, where the number of *d*'s is given by the precision (default 6).
`g,G`	`double`; use `%e` or `%E` if the exponent is less than -4 or greater than or equal to the precision; otherwise use `%f`. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; pointer (implementation-dependent representation).
`%`	no argument is converted; print a %

A width or precision may be specified as *, in which case the value is computed by converting the next argument (which must be an `int`). For example, to print at most `max` characters from a string `s`,

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

Most of the format conversions have been illustrated in earlier chapters. One exception is the precision as it relates to strings. The following table shows the effect of a variety of specifications in printing ``hello, world'' (12 characters). We have put colons around each field so you can see it extent.

```
:%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     :
```

A warning: `printf` uses its first argument to decide how many arguments follow and what their type is. It will get confused, and you will get wrong answers, if there are not enough arguments of if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */
printf("%s", s);   /* SAFE */
```

The function `sprintf` does the same conversions as `printf` does, but stores the output in a string:

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

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `format` as before, but places the result in `string` instead of the standard output; `string` must be big enough to receive the result.

Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines.

7.3 Variable-length Argument Lists

This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way. Since we are mainly interested in the argument processing, `minprintf` will process the format string and arguments but will call the real `printf` to do the format conversions.

The proper declaration for `printf` is

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

where the declaration `...` means that the number and types of these arguments may vary. The declaration `...` can only appear at the end of an argument list. Our `minprintf` is declared as

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

since we will not return the character count that `printf` does.

The tricky bit is how `minprintf` walks along the argument list when the list doesn't even have a name. The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is called `ap`, for ``argument pointer.'' The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

Each call of `va_arg` returns one argument and steps `ap` to the next; `va_arg` uses a type name to determine what type to return and how big a step to take. Finally, `va_end` does whatever cleanup is necessary. It must be called before the program returns.

These properties form the basis of our simplified `printf`:

```
#include <stdarg.h>

/* minprintf: minimal printf with variable argument list */
void minprintf(char *fmt, ...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* make ap point to 1st unnamed arg */
    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); /* clean up when done */
}
```

Exercise 7-3. Revise `minprintf` to handle more of the other facilities of `printf`.

7.4 Formatted Input - Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

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

`scanf` reads characters from the standard input, interprets them according to the specification in `format`, and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with `printf`, this section is a summary of the most useful features, not an exhaustive list.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On the end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of the standard input:

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

It scans the `string` according to the format in `format` and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width, an optional `h`, `l` or `L` indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is places in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, is specified, is exhausted. This implies that `scanf` will read across boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value semantics of C. Conversion characters are shown in Table 7.2.

Table 7.2: *Basic Scanf Conversions*

Character	Input Data; Argument type
`d`	decimal integer; `int *`
`i`	integer; `int *`. The integer may be in octal (leading `0`) or hexadecimal (leading `0x` or `0X`).
`o`	octal integer (with or without leading zero); `int *`
`u`	unsigned decimal integer; `unsigned int *`
`x`	hexadecimal integer (with or without leading `0x` or `0X`); `int *`
`c`	characters; `char *`. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use `%1s`

s	character string (not quoted); char *, pointing to an array of characters long enough for the string and a terminating '\0' that will be added.
e,f,g	floating-point number with optional sign, optional decimal point and optional exponent; float *
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list.

As a first example, the rudimentary calculator of Chapter 4 can be written with scanf to do the input conversion:

```
#include <stdio.h>

main()  /* rudimentary calculator */
{
    double sum, v;

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

Suppose we want to read input lines that contain dates of the form

```
25 Dec 1988
```

The scanf statement is

```
int day, year;
char monthname[20];

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

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with the scanf statement:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

`scanf` ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with `scanf`. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line); /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line); /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to `scanf` can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by `scanf`.

A final warning: the arguments to `scanf` and `sscanf` *must* be pointers. By far the most common error is writing

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

instead of

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

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of `scanf` analogous to `minprintf` from the previous section.

Exercise 5-5. Rewrite the postfix calculator of Chapter 4 to use `scanf` and/or `sscanf` to do the input and number conversion.

7.5 File Access

The examples so far have all read the standard input and written the standard output, which are automatically defined for a program by the local operating system.

The next step is to write a program that accesses a file that is *not* already connected to the program. One program that illustrates the need for such operations is `cat`, which concatenates a set of named files into the standard output. `cat` is used for printing files on the screen, and as a general-purpose input collector for programs that do not have the capability of accessing files by name. For example, the command

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

prints the contents of the files `x.c` and `y.c` (and nothing else) on the standard output.

The question is how to arrange for the named files to be read - that is, how to connect the external names that a user thinks of to the statements that read the data.

The rules are simple. Before it can be read or written, a file has to be *opened* by the library function `fopen`. `fopen` takes an external name like `x.c` or `y.c`, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from `<stdio.h>` include a structure declaration called `FILE`. The only declaration needed for a file pointer is exemplified by

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

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is defined with a `typedef`. (Details of how `fopen` can be implemented on the UNIX system are given in Section 8.5.)

The call to `fopen` in a program is

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

The first argument of `fopen` is a character string containing the name of the file. The second argument is the *mode*, also a character string, which indicates how one intends to use the file. Allowable modes include read (`"r"`), write (`"w"`), and append (`"a"`). Some systems distinguish between text and binary files; for the latter, a `"b"` must be appended to the mode string.

If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves

them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, `fopen` will return `NULL`. (The error can be identified more precisely; see the discussion of error-handling functions at the end of Section 1 in Appendix B.)

The next thing needed is a way to read or write the file once it is open. `getc` returns the next character from a file; it needs the file pointer to tell it which file.

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

`getc` returns the next character from the stream referred to by `fp`; it returns `EOF` for end of file or error.

`putc` is an output function:

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

`putc` writes the character `c` to the file `fp` and returns the character written, or EOF if an error occurs. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`. Normally `stdin` is connected to the keyboard and `stdout` and `stderr` are connected to the screen, but `stdin` and `stdout` may be redirected to files or pipes as described in Section 7.1.

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follows:

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

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

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

With these preliminaries out of the way, we are now in a position to write the program `cat` to concatenate files. The design is one that has been found convenient for many programs. If there are command-line arguments, they are interpreted as filenames, and processed in order. If there are no arguments, the standard input is processed.

```
#include <stdio.h>

/* cat:  concatenate files, version 1 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *)

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
       while(--argc > 0)
           if ((fp = fopen(*++argv, "r")) == NULL) {
               printf("cat: can't open %s\n, *argv);
               return 1;
           } else {
              filecopy(fp, stdout);
              fclose(fp);
           }
       return 0;
}

 /* filecopy:  copy file ifp to file ofp */
 void filecopy(FILE *ifp, FILE *ofp)
 {
     int c;

     while ((c = getc(ifp)) != EOF)
         putc(c, ofp);
 }
```

The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are constants, however, *not* variables, so it is not possible to assign to them.

The function

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

is the inverse of `fopen`, it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since most operating systems have some limit on the number of files that a program may have open simultaneously, it's a good idea to free the file pointers when they are no longer needed, as we did in `cat`. There is also another reason for `fclose` on an output file - it flushes the buffer in which `putc` is collecting output. `fclose` is called automatically

for each open file when a program terminates normally. (You can close `stdin` and `stdout` if they are not needed. They can also be reassigned by the library function `freopen`.)

7.6 Error Handling - Stderr and Exit

The treatment of errors in `cat` is not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That might be acceptable if the output is going to a screen, but not if it's going into a file or into another program via a pipeline.

To handle this situation better, a second output stream, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` normally appears on the screen even if the standard output is redirected.

Let us revise `cat` to write its error messages on the standard error.

```
#include <stdio.h>

/* cat:  concatenate files, version 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0];  /* program name for errors */

    if (argc == 1 ) /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                        prog, *argv);
                exit(1);
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error writing stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

The program signals errors in two ways. First, the diagnostic output produced by `fprintf` goes to `stderr`, so it finds its way to the screen instead of disappearing down a pipeline or into an output file. We included the program name, from `argv[0]`, in the message, so if this program is used with others, the source of an error is identified.

Second, the program uses the standard library function `exit`, which terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations. `exit` calls `fclose` for each open output file, to flush out any buffered output.

Within `main`, `return` *expr* is equivalent to `exit`(*expr*). `exit` has the advantage that it can be called from other functions, and that calls to it can be found with a pattern-searching program like those in Chapter 5.

The function `ferror` returns non-zero if an error occurred on the stream `fp`.

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

Although output errors are rare, they do occur (for example, if a disk fills up), so a production program should check this as well.

The function `feof(FILE *)` is analogous to `ferror`; it returns non-zero if end of file has occurred on the specified file.

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

We have generally not worried about exit status in our small illustrative programs, but any serious program should take care to return sensible, useful status values.

7.7 Line Input and Output

The standard library provides an input and output routine `fgets` that is similar to the `getline` function that we have used in earlier chapters:

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

`fgets` reads the next input line (including the newline) from file `fp` into the character array `line`; at most `maxline-1` characters will be read. The resulting line is terminated with `'\0'`. Normally `fgets` returns `line`; on end of file or error it returns `NULL`. (Our `getline` returns the line length, which is a

more useful value; zero means end of file.)

For output, the function `fputs` writes a string (which need not contain a newline) to a file:

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

It returns `EOF` if an error occurs, and non-negative otherwise.

The library functions `gets` and `puts` are similar to `fgets` and `fputs`, but operate on `stdin` and `stdout`. Confusingly, `gets` deletes the terminating `'\n'`, and `puts` adds it.

To show that there is nothing special about functions like `fgets` and `fputs`, here they are, copied from the standard library on our system:

```
/* fgets:  get at most n chars from 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:  put string s on file iop */
int fputs(char *s, FILE *iop)
{
    int c;

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

For no obvious reason, the standard specifies different return values for `ferror` and `fputs`.

It is easy to implement our `getline` from `fgets`:

```
/* getline:  read a line, return length */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```

Exercise 7-6. Write a program to compare two files, printing the first line where they differ.

Exercise 7-7. Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found?

Exercise 7-8. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.

7.8 Miscellaneous Functions

The standard library provides a wide variety of functions. This section is a brief synopsis of the most useful. More details and many other functions can be found in Appendix B.

7.8.1 String Operations

We have already mentioned the string functions `strlen`, `strcpy`, `strcat`, and `strcmp`, found in `<string.h>`. In the following, `s` and `t` are `char *`'s, and `c` and `n` are `int`s.

`strcat(s,t)`	concatenate `t` to end of `s`
`strncat(s,t,n)`	concatenate `n` characters of `t` to end of `s`
`strcmp(s,t)`	return negative, zero, or positive for `s < t`, `s == t`, `s > t`
`strncmp(s,t,n)`	same as `strcmp` but only in first `n` characters
`strcpy(s,t)`	copy `t` to `s`
`strncpy(s,t,n)`	copy at most `n` characters of `t` to `s`
`strlen(s)`	return length of `s`
`strchr(s,c)`	return pointer to first `c` in `s`, or `NULL` if not present
`strrchr(s,c)`	return pointer to last `c` in `s`, or `NULL` if not present

7.8.2 Character Class Testing and Conversion

Several functions from `<ctype.h>` perform character tests and conversions. In the following, `c` is an `int` that can be represented as an `unsigned char` or `EOF`. The function returns `int`.

`isalpha(c)` non-zero if `c` is alphabetic, 0 if not
`isupper(c)` non-zero if `c` is upper case, 0 if not
`islower(c)` non-zero if `c` is lower case, 0 if not
`isdigit(c)` non-zero if `c` is digit, 0 if not
`isalnum(c)` non-zero if `isalpha(c)` or `isdigit(c)`, 0 if not
`isspace(c)` non-zero if `c` is blank, tab, newline, return, formfeed, vertical tab
`toupper(c)` return `c` converted to upper case
`tolower(c)` return `c` converted to lower case

7.8.3 Ungetc

The standard library provides a rather restricted version of the function `ungetch` that we wrote in Chapter 4; it is called `ungetc`.

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

pushes the character `c` back onto file `fp`, and returns either `c`, or `EOF` for an error. Only one character of pushback is guaranteed per file. `ungetc` may be used with any of the input functions like `scanf`, `getc`, or `getchar`.

7.8.4 Command Execution

The function `system(char *s)` executes the command contained in the character string `s`, then resumes execution of the current program. The contents of `s` depend strongly on the local operating system. As a trivial example, on UNIX systems, the statement

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

causes the program `date` to be run; it prints the date and time of day on the standard output. `system` returns a system-dependent integer status from the command executed. In the UNIX system, the status return is the value returned by `exit`.

7.8.5 Storage Management

The functions `malloc` and `calloc` obtain blocks of memory dynamically.

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

returns a pointer to `n` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied.

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

returns a pointer to enough free space for an array of `n` objects of the specified size, or `NULL` if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by `malloc` or `calloc` has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

```
int *ip;

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

`free(p)` frees the space pointed to by `p`, where `p` was originally obtained by a call to `malloc` or `calloc`. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling `malloc` or `calloc`.

It is also an error to use something after it has been freed. A typical but incorrect piece of code is this loop that frees items from a list:

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

The right way is to save whatever is needed before freeing:

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

Section 8.7 shows the implementation of a storage allocator like `malloc`, in which allocated blocks may be freed in any order.

7.8.6 Mathematical Functions

There are more than twenty mathematical functions declared in `<math.h>`; here are some of the more

frequently used. Each takes one or two `double` arguments and returns a `double`.

`sin(x)`	sine of *x*, *x* in radians
`cos(x)`	cosine of *x*, *x* in radians
`atan2(y,x)`	arctangent of *y*/*x*, in radians
`exp(x)`	exponential function e^x
`log(x)`	natural (base *e*) logarithm of *x* *(x>0)*
`log10(x)`	common (base 10) logarithm of *x* *(x>0)*
`pow(x,y)`	x^y
`sqrt(x)`	square root of *x* *(x>0)*
`fabs(x)`	absolute value of *x*

7.8.7 Random Number generation

The function `rand()` computes a sequence of pseudo-random integers in the range zero to `RAND_MAX`, which is defined in `<stdlib.h>`. One way to produce random floating-point numbers greater than or equal to zero but less than one is

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

(If your library already provides a function for floating-point random numbers, it is likely to have better statistical properties than this one.)

The function `srand(unsigned)` sets the seed for `rand`. The portable implementation of `rand` and `srand` suggested by the standard appears in Section 2.7.

Exercise 7-9. Functions like `isupper` can be implemented to save space or to save time. Explore both possibilities.

Back to Chapter 6 -- Index -- Chapter 8
