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