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

Chapter 8 - The UNIX System Interface

The UNIX operating system provides its services through a set of *system calls*, which are in effect functions within the operating system that may be called by user programs. This chapter describes how to use some of the most important system calls from C programs. If you use UNIX, this should be directly helpful, for it is sometimes necessary to employ system calls for maximum efficiency, or to access some facility that is not in the library. Even if you use C on a different operating system, however, you should be able to glean insight into C programming from studying these examples; although details vary, similar code will be found on any system. Since the ANSI C library is in many cases modeled on UNIX facilities, this code may help your understanding of the library as well.

This chapter is divided into three major parts: input/output, file system, and storage allocation. The first two parts assume a modest familiarity with the external characteristics of UNIX systems.

Chapter 7 was concerned with an input/output interface that is uniform across operating systems. On any particular system the routines of the standard library have to be written in terms of the facilities provided by the host system. In the next few sections we will describe the UNIX system calls for input and output, and show how parts of the standard library can be implemented with them.

8.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even keyboard and screen, are files in the file system. This means that a single homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before you read and write a file, you must inform the system of your intent to do so, a process called *opening* the file. If you are going to write on a file it may also be necessary to create it or to discard its previous contents. The system checks your right to do so (Does the file exist? Do you have permission to access it?) and if all is well, returns to the program a small non-negative integer called a *file descriptor*. Whenever input or output is to be done on the file, the file descriptor is used instead of the name to identify the file. (A file descriptor is analogous to the file pointer used by the standard library, or to the file handle of MS-DOS.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

Since input and output involving keyboard and screen is so common, special arrangements exist to make this convenient. When the command interpreter (the ``shell'') runs a program, three files are open, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error. If a program reads 0 and writes 1 and 2, it can do input and output without worrying about opening files.

The user of a program can redirect I/O to and from files with < and >:

```
prog <infile >outfile
```

In this case, the shell changes the default assignments for the file descriptors 0 and 1 to the named files. Normally file descriptor 2 remains attached to the screen, so error messages can go there. Similar observations hold for input or output associated with a pipe. In all cases, the file assignments are changed by the shell, not by the program. The program does not know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

8.2 Low Level I/O - Read and Write

Input and output uses the `read` and `write` system calls, which are accessed from C programs through two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a character array in your program where the data is to go to or to come from. The third argument is the number is the number of bytes to be transferred.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Each call returns a count of the number of bytes transferred. On reading, the number of bytes returned may be less than the number requested. A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the return value is the number of bytes written; an error has occurred if this isn't equal to the number requested.

Any number of bytes can be read or written in one call. The most common values are 1, which means one character at a time (``unbuffered''), and a number like 1024 or 4096 that corresponds to a physical block size on a peripheral device. Larger sizes will be more efficient because fewer system calls will be made.

Putting these facts together, we can write a simple program to copy its input to its output, the equivalent of the file copying program written for Chapter 1. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#include "syscalls.h"

main()  /* copy input to output */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

We have collected function prototypes for the system calls into a file called `syscalls.h` so we can include it in the programs of this chapter. This name is not standard, however.

The parameter `BUFSIZ` is also defined in `syscalls.h`; its value is a good size for the local system. If the file size is not a multiple of `BUFSIZ`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher-level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` that does unbuffered input, by reading the standard input one character at a time.

```
#include "syscalls.h"

/* getchar:  unbuffered single character input */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

`c` must be a `char`, because `read` needs a character pointer. Casting `c` to `unsigned char` in the return statement eliminates any problem of sign extension.

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.