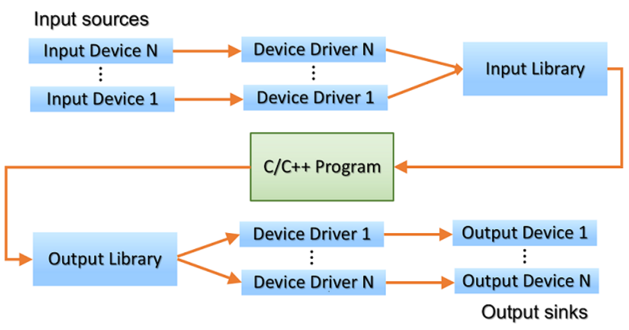
Input/Output Facilities in C

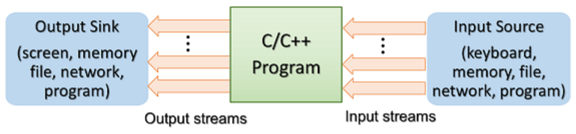
Introduction

A program without data or input/output facilities is not of much use. To make computing useful, a program must be able to read data from a variety of input devices and write computation results to a variety of output devices. In general, programmers require a way to separate their programs from the internal working of input and output devices. It is burdensome and expensive for programmers to implement facilities for their programs to interact with a bewildering range of I/O devices. In addition, these programs have to be continually updated and upgraded to interact with new I/O devices. Modern operating systems understand the need for separation between application programs and I/O devices. They require designers of I/O devices to implement specialized programs called device drivers that interact with an application-programming interface presented by the operating system. When a program requires data from an input device, it will call an operating system procedure that in turn will call a function implemented by the device's driver. Since the designers of the device implemented its driver function, it will know how to read data from the device. Then the data read from the device is passed up to the operating system which then transfers the data to the program. This over simplified description reveals an additional problem. Rather than embedding low-level mechanisms to interact with I/O devices, programmers must now interact with the operating system. To make programs portable across different platforms, the programming language provides an I/O library that abstracts away the details of individual operating system procedures that interact with specific devices. The abstraction provided by an I/O library so that programmers don't have to think in terms of operating system procedures and low-level details of device drivers and devices is illustrated in the following figure.



Streams

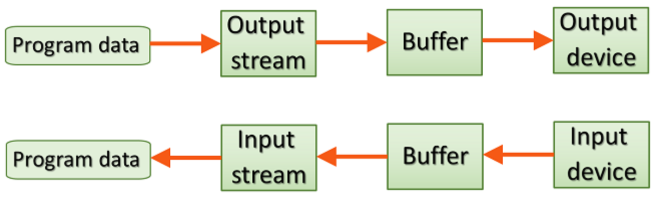
The I/O facilities provided by the C/C++ standard libraries is based on the concept of a *stream*. A stream is an abstraction for sequences of bytes consumed by programs as input and sequences of bytes generated by programs as output. A stream can be a file or some other source or consumer of data including processes and peripheral devices such as keyboards and display screens. The following figure illustrates the stream model.


Output Sink
(screen, memory file, network, program)
C/C++ Program
Input Source
(keyboard, memory, file, network, program)
Output streams
Input streams

Simple programs obtain their input from the keyboard and write their output to the screen. Larger programs require additional streams. For example, the program can communicate with other programs or obtain data from the network as a sequence of bytes or write data as a sequence of bytes to a file stored in secondary storage devices. Thus, stream I/O constitutes a flexible communication path between programs and I/O devices including other programs.

`FILE` type

The I/O facilities of the C standard library are presented in header file `<stdio.h>`. Since a stream represents a "flow" of bytes arriving from or to devices, it must provide a memory buffer modeled as a data queue to facilitate data transfer between programs and devices. The idea of buffering streams is illustrated in the following figure.



In addition to buffers, a stream must provide control information that provides the current position within the stream, pointers to any associated buffers, and indications of whether an error or end of file has occurred. The data type `FILE` defined in `<stdio.h>` encapsulates information about a stream: the location of buffers, current byte position within the stream, whether the stream is being written to or being read from, whether errors or end of file have occurred.

Text and Binary Streams

There are two general forms of streams: *text* and *binary*. A text stream consists of a sequence of characters divided into lines, each line consisting of zero or more characters followed by a line break. The standard library functions translate between the underlying platform's line break representation and the newline character `'\n'` used by C programs. The standard requires implementations to support text stream lines of *at least* 254 characters including the newline. Text streams are portable and human-readable when they consist of complete lines made from printable characters encoded in a standard character set such as ASCII or UTF-8. Characters may have to be altered to conform to differing conventions for representing text in the platform reading text from a stream. As a consequence, data read to or written from a text stream will not necessarily compare equal to the stream's byte content.

```c
char *file_name;

// initialize file_name

// open the file as an output text stream
FILE *file = fopen(file_name, "w");
// check to ensure that file is not NULL
fputs("\n", file);
```

For instance, platforms model newlines differently. On DOS/Windows, the above code will write 2 bytes to the file, whereas on Unix/Linux, the same code will write 1 byte. For Unix/Linux operating systems, the newline character is equivalent to the line feed character `'\n'` with an integral value `0x0A`. DOS/Windows systems represent the newline using the carriage return and line feed characters `'\r''\n'` with integral values `0x0D 0x0A`. Older versions of the Apple operating systems represent newline using the carriage return character `'\r'`. Confusingly, all this means that a program executing in Unix/Linux platforms will count an extra character per line for a text file created in Windows while the same program executing in Windows will count the carriage return and line feed characters as a single newline character.

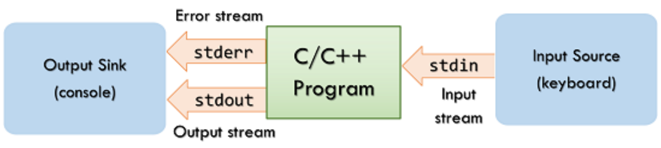
Binary streams are sequences of data values of type `char`. Because any C data value can be mapped onto an array of values of type `char`, binary streams can transparently record internal program data such as integers and floating-point numbers. The following code fragment opens a file as a binary stream. Regardless of the platform, the code will write exactly 1 byte for a newline.

```c
char *file_name;

// initialize file_name

// open the file as an output binary stream
FILE *file = fopen(file_name, "wb");
// check to ensure that file is not NULL
fputs("\n", file);
```

In general, binary files are non-portable unless the source or destination processing units take into account the data's endianess.

Standard I/O Streams

When a C program begins execution, the standard library provides three standard streams of type `FILE*` for the program to interact with its environment: *standard input stream* having identifier `stdin` represents an abstraction of the physical flow of characters from an input device to the program; *standard output stream* having identifier `stdout` represents the abstraction of the physical flow of characters from the program to an output device; *standard error stream* having identifier `stderror` represents an abstraction for the program to write error or diagnostic messages to an output device. The conceptual model of these three streams is illustrated in the following figure.



These three standard streams have *static extent* (lifetime throughout the program's duration) and *external linkage* (public across translation units). By default, `stdin` provides a communication channel for characters from the physical keyboard device to be read by the program; `stdout` and `stderr` provide a communication channel for the console to display characters written by the program.

The expressions `stdin`, `stdout`, and `stderr` are often defined as addresses of static or global stream descriptors:

```c
extern FILE __iob[FOPEN_MAX];

#define stdin  (&__iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

This program illustrates the use of `stdin`, `stdout`, and `stderr` streams to read from the keyboard and print messages from the program to the console:

```c
#include <stdio.h>

int main(void) {
  FILE *psi = stdin, *pso = stdout, *pse = stderr;
  fputs("Enter a string: ", pso);
  char str[255];
  fgets(str, 255, psi);
  fputs("You entered: ", pso);
  fputs(str, pso);
  fputs("There are no error messages", pse);
  return 0;
}
```

Simplest character I/O functions

Functions `getchar` and `putchar` are presented in `<stdio.h>` as

```c
int getchar(void);
void putchar(int);
```

`getchar` returns the next input character it reads from `stdin` each time it is called or the macro `EOF` when there are no more characters present. `EOF` is declared in `<stdio.h>` as a negative integral constant expression to signal the exhaustion of input data. `putchar` writes a character to `stdout` and returns the written character or `EOF` if an error occurs. Because `EOF` is dually used by `getchar` to signal end of file and by `putchar` to signal write errors, it is better to use function `feof` to determine whether end of file has indeed been encountered when `EOF` is returned. The following program echoes to the console a character typed on the keyboard followed by the newline:

```c
int main(void) {
  putchar('e'); putchar('n'); putchar('t'); putchar('e');
  putchar('r'); putchar(':'); putchar(' ');
  int ch = getchar();
  if (ch != EOF) {
    putchar(ch);
  }
  return 0;
}
```

While functions `getchar` and `putchar` read and write a single character, functions `fgets` and `fputs` read and write a sequence of characters, respectively:

```c
char *fgets(char *string, int n, FILE *stream);
int fputs(char const* string, FILE *stream);
```

`fgets` reads at most `n-1` characters into array `string` from `stream` stopping if newline is encountered; the newline is included in the array, which is terminated by `\0`. The function returns `string` if end of file is encountered or `NULL` if there is a read error.

`fputs` writes the characters in the array `string` (not including the null terminator `\0`) to `stream`. The function returns `EOF` if a write error occurs; otherwise, a non-negative value.

The following program illustrates use of `fputs` to write to `stdout` and `fgets` to read from `stdout`:

```c
#include <stdio.h>
enum { STR_SIZE = 255 };
int main(void) {
  char str[STR_SIZE];
  fputs("Enter a string: ", stdout);
  fgets(str, STR_SIZE, stdin);
  fputs("You entered: ", stdout);
  fputs(str, stdout);
  return 0;
}
```

Redirection of `stdout` and `stderr` streams

Interactive shells allow standard streams `stdin`, `stdout`, and `stderr` to be redirected from their default meanings. For `stdout`, redirection means that a program will write characters to a file rather than display them on the console. To substitute the destination stream for `stdout` from the console to a file, use the output stream redirection operator `>`. If a program `a.exe` will write characters to a file `output-file` rather than `stdout` using the command `./a.exe >output-file` in the interactive shell.

A *file descriptor* (which is a number that the operating system uses as a handle to an open file) can be nominally added to the output stream direction operator. The standard streams `stdin`, `stdout`, and `stderr` are assigned file descriptors `0`, `1`, and `2`, respectively by the operating system. The command `./a.exe 1>output-file` will redirect program `a.exe`'s output from `stdout` to file `output-file`.

Suppose we compile and link the following source into an executable program `a.exe`:

```c
#include <stdio.h>
int main(void) {
  fputs("This is going to stdout\n", stdout);
  fputs("This is going to stderr\n", stderr);
  return 0;
}
```

The following lines are displayed on the console when program `a.exe` is executed:

```
This is going to stdout
This is going to stderr

```

Since the characters written to `stdout` and `stderr` streams are displayed on the console, it is hard to tell what is an error or diagnostic message and what is normal program output. Using the output redirection operators, the operating system can be made to redirect these streams to different destinations.

The command `./a.exe >out.txt` will redirect `stdout` to a file `out.txt` with the following sequence of characters written to `stderr`:

```
This is going to stderr

```

To redirect `stderr`, we need to explicitly specify the standard error stream's file descriptor `2` along with the output stream redirection operator, as in `./a.exe 2>err.txt`. While `stderr` is redirected to file `err.txt`, the following sequences of characters are written to `stdout`:

```
This is going to stdout

```

Confirm that `stderr` has been redirected to file `err.txt` by running the command `cat err.txt` in the interactive shell.

The command `./a.exe 1>out.txt 2>err.txt` will redirect `stdout` to file `out.txt` and redirect `stderr` to file `err.txt`.

The command `/a.exe 1>both.txt 2>&1` will redirect both `stdout` and `stderr` to the same file `both.txt`. Or, use this simpler syntax: `./a.exe &>both.txt`.

Using the output redirection operator `>` causes an empty file to be created. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

To append at the end of an existing file, use the append operator `>>`. If the file doesn't exist, an empty file is created.

Redirection of `stdin` stream

For `stdin`, redirection means that a program will read characters from a file rather than characters typed on the keyboard device. To substitute the source for `stdin` from the keyboard device to a file, use the input stream redirection operator `<`. For example, a program `a.exe` can be made to read characters from a file `input-file` rather than `stdin` using the command `./a.exe <input-file` in the interactive shell. The operating system assigns a file descriptor value `0` to `stdin`. The file descriptor of `stdin` can be explicitly specified as `./a.exe 0<input-file`.

Suppose we compile and link the following source file into an executable program `a.exe`:

```c
#include <stdio.h>
enum { STR_SIZE = 255 };
int main(void) {
  char str[STR_SIZE];
  fgets(str, STR_SIZE, stdin);
  fputs("You entered: ", stdout);
  fputs(str, stdout);
  return 0;
}
```

If an input file `in.txt` contains the following line of text:

```
today is a good day
```

then the command `./a.exe 0<in.txt` in the interactive shell will write the following sequences of characters to `stdout`:

```
You entered: today is a good day
```

Both `stdin` and `stdout` can be redirected with the command `./a.exe 0<in.txt 1>out.txt`.

File copy: Version 1 and `EOF`

Given the simplest character-by-character I/O functions `getchar` and `putchar`, a surprising amount of useful code can be written without knowing anything more about input and output. A simple example is a program that copies characters from its `stdin` stream to its `stdout` stream. This program can behave more generally as a file copy command by redirecting the program's `stdin` to a source file and by redirecting `stdout` to a destination file.

```c
#include <stdio.h>

// file copy - version 1: copy from stdin stream to stdout stream.
// to create a copy of a source file in an other file, use the input
// and output redirection operator in the shell
void filecopy(void) {
  char ch;
  while ((ch = getchar()) != EOF) {
    putchar(ch);
  }
}

int main(void) {
  filecopy();
  return 0;
}
```

However, the program doesn't work as expected with binary files such as pdf files. To understand the bug in the above code fragment, we need to understand two issues. The standard allows implementations to choose the definition of type `char` to have the same range, representation, and behavior as either `signed char` or `unsigned char`. This means that `char` is a separate type from the other two and is not compatible with either. Macro `CHAR_MIN`, defined in `<limits.h>`, will have one of the values `0` or `SCHAR_MIN`, and this can be used to determine whether the implementation has chosen to represent type `char` as `signed char` or `unsigned char`:

```c
#include <limits.h>

// return 0 if char is signed char; return 1 if char is unsigned char
int char_type(void) {
  return (CHAR_MIN == 0) ? 1 : 0;
}
```

We can also write a more general macro `IS_NEG_TYPE` to determine whether a type is `signed` or `unsigned`. The method relies on the idea that `unsigned` types can never contain negative values. So, we cast a value represented by all bits set to `1` and compare if the value is evaluated as `-1` or `0`:

```c
// macro returns 1 if a type is signed; otherwise 0 is returned
#define IS_NEG_TYPE(type) ((type)(-1) < 0)
```

Since its dependent on a particular implementation whether a `char` is `signed char` or `unsigned char`, the standard library specifies that arguments to character handling functions must be representable as an `unsigned char`. This means that a function such as `getchar` reads `unsigned char` values with range from `0` to `255` from `stdin` since signedness has little meaning in character sets.

The second issue arises from the problem of distinguishing the end of input from valid data when reading an input stream. Functions that read from input streams such as `getchar` must be able to return all of the 256 possible non-negative values in a byte plus a distinctive value when bytes are exhausted in the stream that cannot be confused with any real character in the stream. This distinctive end of file value is specified by macro `EOF`. The standard requires `EOF` to be a negative value with most implementations choosing `-1` as the `EOF` value. The particular numerical value of `EOF` is not important but it has to be a negative value because there is no character in any of the standard character sets that has a negative value. Hence, return values of `getchar` and other input functions that read from input streams are chosen to be `int` because `int` values are big enough to hold `EOF` in addition to any possible non-negative character value ranging from 0 to 255. To make all of this work, input functions read bytes from input streams as `unsigned char`s and return values between `0` and `255`. When the characters in the input stream are exhausted, input functions will signal this state by returning `EOF`.

From this discussion, you can surmise that the bug is caused by assigning the return value of `getchar` to a `char` object. This bug doesn't cause problems for text streams. However, binary streams store a sequence of `unsigned char` values and it is possible to read a byte having integral value between 128 and 255 which is interpreted by the return `char` type as a negative value equivalent to `EOF`. This means that the file copy program can prematurely return `EOF` without reading the entire sequence of bytes in the binary stream.

File copy: Version 2

Version 2 of the file copy program eliminates a premature return before the input stream is exhausted of bytes by declaring variable `ch` as type `int`.

```c
#include <stdio.h>

// file copy - version 2: copy from stdin stream to stdout stream
// returns 0 if there's no read error; otherwise non-zero value
int filecopy(void) {
  // change type specifier of variable ch from char to int
  int ch;
  // read bytes till end of file or until first read error
  while ((ch = getchar()) != EOF) {
    putchar(ch);
  }
  return 0; // indicate success
}

int main(void) {
  filecopy();
  return 0;
}
```

Function `feof` returns a non-zero value if the end of file indicator associated with the stream is set. Checking the return value of `feof` in a `while` loop, as in:

```c
int ch;
while ( (ch = getchar()) != feof()  {
  putchar(ch);
}
```

will cause an infinite loop when a read error happens, but end of file has not been reached yet.

Error functions

C doesn't have the C++ exception mechanism, so you need to check for errors after calling library functions. Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression `errno` (declared in `<errno.h>`) may contain an error number that gives further information about the most recent error. The descriptions of the errors listed in `<errno.h>` can be viewed here.

```c
// declared in <stdio.h>

// clears the end of file and error indicators for stream
void clearerr(FILE *stream);
// returns non-zero if the end of file indicator for stream is set
int feof(FILE *stream);
// returns non-zero if the error indicator for stream is set
int ferror(FILE *stream);
/*
prints to stderr both s and an implementation-defined error message
corresponding to the integer in errno, as if
  fprintf(stderr, "%s: %s\n", s, strerror(errno));
*/
void perror(char const *s);

// declared in <string.h>

// return pointer to implementation-defined string corresponding to error n
char *strerror(n);
```

Character-by-character output functions

The standard library declares the following functions in `<stdio.h>` to *write* characters and sequences of characters to an output stream:

```c
// these two functions read from stdout
int putchar(int ch); // may also be implemented as a macro
int puts(char const *string);

// these 3 functions require the caller to specify the output stream
int putc(int ch, FILE *stream); // may also be implemented as a macro
int fputc(int ch, FILE *stream);
int fputs(char const *string, FILE *stream);
```

Functions `putchar`, `putc`, and `fputc` can be used interchangeably:

```c
int ch;
// all 3 functions write the character ch to stdout
putchar(ch);
putc(ch, stdout);
fputc(ch, stdout);
```

Function `puts` writes characters in `string` *plus* a newline to `stdout` while function `fputs` writes characters in `string` to `stream`. Neither function will write the zero terminating value `'\0'` in `string`. Both return `EOF` if error occurs, otherwise a non-negative value. Recall C-strings are sequences of characters terminated by zero value (ASCII `NUL` has all 8 bits cleared to zero and represented by character literal `'\0'`). The following calls to `puts` and `fputs` write the same sequence of characters into `stdout`:

```c
puts("This is a line of text");
fputs("This is a line of text\n", stdout);
```

There's nothing special about `fputs`, it is easily implemented using a character write function:

```c
// write string s in stream is
// return EOF if error occurs, otherwise a non-negative value (0)
int fputs(char const *str, FILE *is) {
  int c;
  // continue to read a character from str and write it into the stream
  // until the zero terminating character is reached
  while (c = *str++) {
    putc(c, is);
  }
  return ferror(is) ? EOF : 0;
}
```

Character-by-character input functions

The standard library declares the following functions in `<stdio.h>` to *read* characters and sequences of characters from an input stream:

```c
// these 2 functions read from stdin
int getchar(void); // may also be implemented as a macro
char *gets(char *buffer);

// these 3 functions require the caller to specify the input stream
int getc(FILE *stream); // may also be implemented as a macro
int fgetc(FILE *stream);
char *fgets(char *buffer, int n, FILE *stream);
```

Functions `getchar`, `getc`, and `fgetc` can be used interchangeably:

```c
// all 3 functions write the character ch to stdout
int ch = getchar();
ch = getc(stdin);
ch = fgetc(stdin);
```

Function `gets` reads the next line from `stdin` into array `buffer`; it discards the terminating newline character and writes the null character `'\0'` immediately after the last character written into `buffer`. The function returns `buffer` or `NULL` if end of file or error occurs.

Never use `gets`

This is good advice because `gets` provides no way to control how much data is read into a buffer from `stdin`. The following code fragment assumes that `gets` will not read more than `BUFFER_SIZE - 1` characters from `stdin`. This is an invalid assumption, and the resulting operation can result in a buffer overflow. A buffer overflow occurs if a program attempts to write more data in a buffer than it can hold or write into a memory area outside the boundaries of the buffer.

```c
#include <stdio.h>

enum { BUFFER_SIZE = 1024 };

void func(void) {
  char buf[BUFFER_SIZE];
  if (gets(buf) == NULL) {
    // handle error
  }
  // there may have been out-of-bounds writes by gets ...
}
```

Because `gets` is inherently unsafe and should never be used, it has been deprecated in ISO C99 and removed from ISO C11.

Use `fgets` as replacement for `gets`

`char *fgets(char *buffer, int n, FILE *stream);` is the compliant alternative to `gets`. It reads, at most, `n-1` number of characters from `stream` into an array `buffer`, stopping if the newline is encountered; the newline is included in `buffer` which is terminated by null character `'\0'`. The function returns `buffer` or `NULL` if end of file or read error occurs. This solution is compliant because the number of characters copied from `stream` to `buffer` cannot exceed the allocated memory. There's nothing special about `fgets`, it is easily implemented using a character read function:

```c
// read at most n-1 characters, stopping if the newline is encountered
char *fgets(char *buffer, int n, FILE *is) {
  char *cb = buffer;
  int ch;
  // read characters from stream until n-1 characters have been
  // read or there are no more characters to be read
  while (--n > 0 && (ch = getc(is)) != EOF) {
    if ((*cb++ = ch) == '\n') {
      break;
    }
  }
  *cb = '\0'; // null-terminate

  // return NULL if there was a read error
  return ((ch == EOF && cb == buffer) || ferror(is)) ? NULL : buffer;
}
```

File copy: Version 3

Rather than reading and writing character-by-character, functions `fgets` and `fputs` can be used to read and write entire lines of text:

```c
#include <stdio.h>

// file copy - version 3: copy from src stream to dest stream
// returns 0 if there's no read error; otherwise non-zero value
enum { STR_BUFFER = 1024 };
int filecopy(FILE *src, FILE *dest) {
  char buff[STR_BUFFER];
  // read lines till end of file or until first read error
  while (fgets(buff, STR_BUFFER, src)) {
    fputs(buff, dest);
  }

  // ferror(stream) returns non-zero value if error indicator associated
  // with stream is set; otherwise zero value is returned
  return ferror(src) || ferror(dest); // 0 for success and 1 for failure
}

int main(void) {
  filecopy(stdin, stdout);
  return 0;
}
```

Function-like macros

Since macros are more efficient than functions, certain implementations may also provide macro-like functions for `putchar`, `putc`, `getchar`, and `getc`:

```c
#define putchar(ch)       fputc(ch, stdout)
#define putc(ch, stream) fputc(ch, stream)
#define getchar()         fgetc(stdin)
#define getc(stream)      fgetc(stream)
```

It may be strange that a function could be implemented as both a function and a macro, but the standard says that any function declared in a header may be additionally defined as a function-like macro in the header. The library insists on a function definition even though a corresponding function-like macro exists for those cases where a program wishes to pass a function's address as a parameter to another function. How can we define a function called `getchar` if that name is also recognized by the preprocessor? This is possible if the function definition encloses the function name in parenthesis:

```c
int (getchar)(void) {
  // provide code here to read a character byte from stdin
}

// now &getchar will be defined if needed
```

Read these C++ faq notes from C++ experts to understand their view that macros are evil. One particular problem with macros is that arguments can be evaluated multiple times:

```c
// SQUARE is a function-like macro
#define SQUARE(x) ((x) * (x))

// definition of function get_int that returns an int
int get_int(void) { return 1; }

/*
the expansion of the function-like macro by the preprocessor will
call function get_int() twice ...
In contrast, if SQUARE was a function, its argument will be evaluated
once and no more ...
*/
int x = SQUARE(get_int());
```

Most implementations promise that their function-like macros for `putchar` and `putc` will not evaluate the stream multiple times. However, if that is not the case, in the following code fragment, `putc` might call a user-defined function `get_file` to return a stream (`FILE*`) more than once with potentially harmful side effects. In contrast, the argument `get_file()` in the call to `fputc` will be evaluated only once to initialize the stream parameter of `fputc`.

```c
FILE* get_file(void) { ... }

// ... other code here ...

putc('z', get_file());
fputc('y', get_file());
```

Opening and closing streams (files)

The examples so far have read from `stdin` and written to `stdout` streams, which are automatically defined for a program by the operating system. We've used the input and output stream redirection operators to redirect `stdin` and `stdout` to data files. The next step is to allow programs to directly access files that are not already connected to the program. C handles file I/O similar to standard I/O. To directly read from a file, you construct a `FILE` stream that connects to an input file, and use functions such as `getchar` and `fgets`. Similarly, to write to a file, you construct a `FILE` stream that connects to the output file, and use functions such as `putchar` and `fputs`. The steps are:

1. Connect a stream to a file (i.e., open the file) using function `fopen`.
2. Read from or write to a file using functions such as `getchar`, `fgets`, `putchar`, and `fputs`.
3. Disconnect the file from the stream (i.e., close the file) using function `fclose`.

The functions `fopen` and `fclose` have the following declarations in `<stdio.h>`:

```c
FILE* fopen(char const *name, char const *mode);
int   fclose(FILE *stream);
```

The function `fopen` takes two C-string arguments representing a file name and a mode. The file name is used to open or create a file and associate it with a stream. A pointer of type `FILE*` is returned to identify the stream for other I/O operations. If any error is detected during opening of the stream, `fopen` stores an error code into `errno` and returns a null pointer.

The function `fclose` is the inverse of `fopen`: it breaks the connection between the file pointer and the external name that was established by `fopen`. Since most operating systems have a 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're no longer needed. A second reason for calling `fclose` on an output stream is that it flushes the internal data buffer in which a write function such as `fputc` has been collecting output. Note that `fclose` is called automatically for each open file when the program terminates normally. Programmers can also close `stdin` and `stdout` if they're not needed. They can also be reassigned by the library function `freopen`. `fclose` returns `EOF` if an error is detected; otherwise it returns zero.

The second C-string argument to `fopen` is the mode specifying how the file is intended to be used. The allowable modes are listed in the following table:

Mode	Meaning
`"r"`	Open an existing file input
`"w"`	Create a new file or truncate an existing one for output
`"a"`	Create a new file or append to an existing one for output
`"r+"`	Open an existing file for update (both reading and writing) starting at the beginning of the file
`"w+"`	Create a new file or truncate an existing one for update
`"a+"`	Create a new file or append to an existing one for update

The standard allows any of the types listed in the table to be followed by the character `b` to indicate "binary" (as opposed to "text") stream is to be created. The distinction under Unix/Linux was blurred because both kinds of files are handled the same, other operating systems are not so lucky. The following table lists some properties of each of the stream modes:

Property	`r`	`w`	`a`	`r+`	`w+`	`a+`
Named file must already exist	yes	no	no	yes	no	no
Existing file's contents are lost	no	yes	no	no	yes	no
Read from stream permitted	yes	no	no	yes	yes	yes
Write to stream permitted	no	yes	yes	yes	yes	yes
Write begins at end of stream	no	no	yes	no	no	yes

Consider functions `open_stream` and `close_stream` that open and close normal text files. They handle error conditions and print diagnostics as necessary, and their return values match those of `fopen` and `fclose`:

```c
#include <errno.h>  // for errno
#include <string.h> // for strerro
#include <stdio.h>  // FILE*, fopen, fprintf, fclose, perror

// Open filename for input or output based on mode argument;
// return NULL if problem
```

```c
FILE *open_stream(char const *filename, char const *mode) {
  errno = 0; // clear previous errors
  // functions below may choke on NULL filename
  filename = (filename == NULL) ? "\0" : filename;
  FILE *stream = fopen(filename, mode);
  // if fopen fails, we call perror to display reason
  if (stream == NULL) {
    char log[128];
    sprintf(log, "open_stream(\"%s\") failed", filename);
    perror(log);
  }
  return stream;
}

// close stream
int close_stream(FILE *stream) {
  if (stream == NULL) {
    return 0;
  }

  errno = 0; // clear previous errors
  int cv = fclose(stream);
  if (cv == EOF) {
    perror("closing of stream failed");
  }
  return cv;
}
```

Reading and writing text streams

This example displays the contents of a text file. Notice we read an entire line or up to `BUFF_SIZE-1` characters using `fgets` rather than reading character-by-character using `getchar`.

```c
// display the contents of a text file to stdout
#include <errno.h>
#include <string.h>
#include <stdio.h>

// forward declarations of functions defined above
FILE *open_stream(char const *filename, char const *mode);
int close_stream(FILE *stream);

enum { BUFF_SIZE = 256 };

int main(int argc, char ** argv) {
  if (argc == 1) {
    printf( "Usage ./%s <filename>\n", argv[0] );
    return 1;
  }

  FILE *stream = open_stream(argv[1], "r"); // text mode
  if (!stream) {
    return 1; // failure
  }
  errno = 0; // clear any previous errors
  char buffer[BUFF_SIZE];
  while (fgets(buffer, BUFF_SIZE, stream)) {
    fputs(buffer, stdout);
  };

  // was there a read error?
  if (!feof(stream)) {
    char log[256];
    // snprintf securely writes - buffer overflows are avoided
    snprintf(log, 256, "error while reading from file \"%s\" failed",
argv[1]);
    perror(log);
    return 1; // failure
  }
  close_stream(stream);

  return 0; // success
}
```

The next example is equivalent to the type command in the Windows Command shell or the cat command in Unix/Linux shells. These two commands concatenate a set of named files to `stdout`. They're used for printing files on the screen, concatenating files, and creating files.

```c
// concatenate files: cat (in Unix/Linux) or type (in Windows)
#include <errno.h>
#include <string.h>
#include <stdio.h>

// forward declarations of functions previously defined
FILE *open_stream(char const *filename, char const *mode);
int close_stream(FILE *stream);

// file copy: copy from src stream to dest stream
// returns 0 if there's no read error; otherwise non-zero value
int filecopy(FILE *src, FILE *dest) {
  int ch;
  while ((ch = getc(src)) != EOF) {
    putc(ch, dest);
  }
  // ferror(stream) returns non-zero value if error indicator
  // associated with stream is set; otherwise zero value is returned
  return ferror(src) || ferror(dest);
}

int main(int argc, char ** argv) {
  int rwerr;

  if (argc == 1) { // no command-line args, copy stdin to stdout
    rwerr = filecopy(stdin, stdout);
  } else {
    while (--argc > 0) {
      FILE *stream = open_stream(*++argv, "r");
      if (!stream) { // unable to open file
        return 1; // failure
      }
      rwerr = filecopy(stream, stdout);
      if (rwerr) { // either read or write error has occurred
        break;
      }
      fclose(stream);
    }
  }
  // if read or write error has occurred, print error message
  if (rwerr) {
    perror("error");
  }
  return 0; // success
}
```

In addition to checking if an error arises when opening a file, notice that this program rigorously checks for read and write errors. Although read and write errors are rare, they do occur (for example, if a disk fills up), so a production program should check for these errors as well.

Reading and writing binary streams

When data is to be read from text files, a program must expend a significant amount of time to convert the input stream of characters into binary integral data represented by C fundamental types such as `short`, `int`, or `long`, the mantissas and exponents for single-precision `float` or double-precision `double` values, and null-terminated character strings. After processing of the data is complete, suppose the program must write the data into a text file. The program must again expend time in converting the internal data format back into a stream of characters for storage in an output text file. In a C program, these conversions are carried out by formatted I/O functions such as `scanf`, `sscanf`, `fscanf`, `printf`, `sprintf`, and `fprintf`.
Many programs produce output files that are used as input files for other programs. If there is no need for humans to read such files, it is a waste of computer resources for the first program to convert its internal data format to a stream of characters, and then for the second program to have to apply an inverse conversion to extract the intended data from the stream of characters. This unnecessary translation can be avoided by using a binary file rather than a text file.

Recall that a text stream consists of a sequence of lines that are terminated by a newline character while a binary stream is an ordered sequence of characters that represent raw data. Functions that read and write sequences of characters such as `fgets` and `puts` are not suitable for reading raw data in binary files because these functions use the newline character as a delimiter and the internal representation of a integral or floating-point number could contain a byte with the same bit pattern as the newline character. Instead, the standard library provides functions `fread` and `fwrite` to perform input and output, respectively on binary files.

```c
size_t fread(void *buffer,
             size_t elem_size, size_t count, FILE *stream);
size_t fwrite(const void *buffer,
              size_t elem_size, size_t count, FILE *stream);
```

In both cases, `stream` is the input or output stream, `buffer` is a pointer to an array of `count` elements, each of which is of size `elem_size` bytes. The actual number of items written is returned by `fwrite`; it will be the same as `count` unless an error occurs. `fread` returns the actual number of *items* read; it may be less than `count` if end of file is encountered or zero if a read error is encountered.

A binary file is created by executing a program that directly stores the internal representation of each data component in the file. For example, the following code creates a binary file called `numbers.bin` containing integer numbers from 11 to 50000 of type `int`:

```c
#include <stdio.h>

enum { MIN_NUM = 11, MAX_NUM = 50000 };

int main(void) {
  FILE *os;
  if ((os = fopen("numbers.bin", "wb")) == NULL) {
    fputs("unable to create numbers.bin\n", stderr);
    return 1;
  }

  for (int i = MIN_NUM; i <= MAX_NUM; ++i) {
    fwrite(&i, sizeof(int), 1, os);
  }
  fclose(os);
  return 0;
}
```

This code reads the previously created binary file and prints the numbers of type `int` to `stdout`:

```c
#include <stdio.h>

int main(void) {
  FILE *is;
  if ((is = fopen("numbers.bin", "rb")) == NULL) {
    fputs("unable to open numbers.bin\n", stderr);
    return 1;
  }

  int i, counter = 1;
  // read int values from input stream until there are no more left
  while (fread(&i, sizeof(int), 1, is)) {
    fprintf(stdout, "%i%s", i, (counter%10) ? " " : "\n");
    ++counter;
  }
  fclose(is);
  return 0;
}
```

Now, consider creating a text file `numbers.txt` to store the same integers:

```c
#include <stdio.h>

enum { MIN_NUM = 11, MAX_NUM = 50000 };

int main(void) {
  FILE *os;
  if ((os = fopen("numbers.txt", "w")) == NULL) {
    fputs("unable to create numbers.txt\n", stderr);
    return 1;
  }

  for (int i = MIN_NUM; i <= MAX_NUM; ++i) {
    fprintf(os, "%i ", i);
  }
  fclose(os);
  return 0;
}
```

Function `fprintf` is similar to `printf` with the only difference being that `fprintf` takes an output stream as an argument while `printf` writes to `stdout` stream. To write the value `12345`, `printf` must first convert the integral value to the C-string `"12345"` and then write six bytes consisting of the ASCII codes for characters `1`, `2`, `3`, `4`, `5`, and null character `\0` to the output stream. Obviously, it will take more time to convert the four byte internal representation of `int` value `12345` to individual characters and then write them to the output stream. Also, the four byte internal representation of value `12345` of type `int` will now require six bytes of storage. A third advantage of using binary files is there is no loss of precision when writing floating-point values. When writing floating-point values to a text file using `fprintf`, the function must convert the floating-point value to a character string whose precision is determined by the placeholder in the format string. A loss of precision will result in this case.

Binary files come with a major disadvantage. A binary file created by a program executing on a big endian CPU will not be correctly read by another program executing on a little endian CPU.

File copy: Version 4

The following program uses `fread` and `fwrite` to implement a fourth version of `filecopy`:

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

enum { STR_BUFFER = 4096 };

int filecopy4(FILE *src, FILE *dst) {
  char buff[STR_BUFFER];
  size_t rb;
  while ((rb = fread(buff, 1, STR_BUFFER, src))) {
    fwrite(buff, 1, rb, dst);
  }
  return ferror(src) || ferror(dst);
}

int main(int argc, char *argv[]) {
  if (argc < 3) {
    fputs("Usage: a.exe src-file dest-file\n", stderr);
    return 0;
  }

  // read and write binary files
  FILE *src = fopen(argv[1], "rb");
  FILE *dst = fopen(argv[2], "wb");
  if (src == NULL || dst == NULL) {
    fputs("Unable to open src and/or dest files\n", stderr);
    return 0;
  }

  clock_t start = clock();
  if (filecopy4(src, dst)) {
    fputs("Unable to copy\n", stderr);
  }
  fprintf(stdout, "Processor time used: %g\n",
                  (clock()-start)/(double)CLOCKS_PER_SEC);
  fclose(src);
  fclose(dst);
  return 0;
}
```

This program will copy both text and binary files. Reading the contents of text files as binary data and writing this binary data directly to the output stream means that there is no formatting and conversion of data by both the reader and writer. Another feature of this program is that it computes and prints the duration of execution of the file copy function. The `clock` function returns a value of type `clock_t` representing the processor time used by the program since execution began. Dividing this value by macro `CLOCKS_PER_SEC` (defined in `<time.h>`) converts it to seconds. To determine how long a particular function is running, `clock` needs to be called twice: once before the call to the function to measure the time taken by the program to execute the code up to the function call and once just after the function returns. The difference between these two times divided by `CLOCKS_PER_SEC` give the function execution time in seconds.

For convenience, file copy function `filecopy3` (that uses line-by-line functions) and `filecopy2` (using character-by-character functions) are listed here:

```c
int filecopy2(FILE *src, FILE *dst) {
  int ch;
  while ((ch = fgetc(src)) != EOF) {
    fputc(ch, dst);
  }
  return ferror(src) || ferror(dst);
}

enum { LINE_BUFFER = 1024 };

int filecopy3(FILE *src, FILE *dst) {
  char buff[LINE_BUFFER];
  while (fgets(buff, LINE_BUFFER, src)) {
    fputs(buff, dst);
  }
  return ferror(src) || ferror(dst);
}
```

Measure and compare the time taken by these three file copy functions using a medium-sized text file and a larger text file.

Reading and writing structures

Consider the structure defined in `person.h`:

```c
#ifndef PERSON_H
#define PERSON_H

struct Person_tag {
  int age;
  float weight;
  char name[24];
};

typedef struct Person_tag Person;

#endif
```

This program writes objects of type `Person` to a binary file:

```c
// binfilew.c - write a binary file (C version)
#include <stdio.h>
#include "person.h"

int main(void) {
  FILE *os = fopen("person.dat", "wb");
  if (os == NULL) {
    fputs("Unable to create person.dat\n", stderr);
    return 1; // error
  }

  Person fs = {42, 231.5f, "Fred Flintstone"};
  fwrite(&fs, sizeof(fs), 1, os);

  Person fs2 = {2, 41.7f, "Pebbles Flintstone"};
  fwrite(&fs2, sizeof(fs2), 1, os);

  Person fs3 = {5, 310.2f, "Dino"};
  fwrite(&fs3, sizeof(fs3), 1, os);
  fclose(os);
  return 0; // success
}
```

We can now read these objects from the file `person.dat` into dynamically allocated memory:

```c
#include <stdio.h>
#include <stdlib.h>
#include "person.h"

int main(void) {
  FILE *is = fopen("person.dat","rb");
  if (is == NULL) {
    fputs("file not found\n", stderr);
    return 1; // failure
  }

  fseek(is, 0L, SEEK_END); // set offset to end of file
  long int offset = ftell(is); // get byte offset from start of file
  int count = offset/sizeof(Person); // number of objects written to file
  Person *pp = malloc(offset);
  if (pp == NULL) {
    fprintf(stderr, "Error allocating %li bytes\n", offset);
    return 1; // failure
  }

  // reset file to first location of file
  fseek(is, 0L, SEEK_SET);
  if (fread(pp, sizeof(Person), count, is) != (size_t)count) {
    fprintf(stderr, "Read error - can't read %d elements\n", count);
    fclose(is);
    return 1; // failure
  }
  fclose(is);

  for (int i = 0; i < count; ++i) {
    fprintf(stdout,"Name: %s\n  Age: %d\n  Weight: %.1f\n",
                   pp[i].name,pp[i].age,pp[i].weight);
  }
  free(pp);
  return 0;
}
```

Functions `fseek` and `ftell` are used to determine the number of objects of type `Person` contained in the binary file. They have the following prototypes:

```c
int fseek(FILE *stream, long int offset, int origin);
long int ftell(FILE *stream);
```

`fseek` allows random access within an open `stream`. The last two arguments specify a position within the file: `offset` is a `long int` that specifies the number of bytes, and `origin` is a seek code indicating from what point in the file `offset` should be measured. The standard defines the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` as values for parameter `origin` to indicate the start of, current position, and end of the file, respectively. For binary files, `ftell` returns the position in the `stream` - the value returned will be the number of characters preceding the current file position.

The following code fragment computes the number of objects of type `Person` stored in file `person.dat`:

```c
is = fopen("person.dat","rb");
fseek(is, 0L, SEEK_END);
offset = ftell(is); // get byte offset from start of file
count = offset/sizeof(Person); // number of objects written to file
```