Chapter - 16
File Input/ Output

I/O Packages

There are 3 different I/O packages available to the C++ programmer:

- The C++ streams package. This package is used for most I/O.

- The unbuffered I/O package. Used primarily for large file I/O and other special operations.

- The C `stdio` package. Many older, C-- programs use this package. (A C-- program is a C program that has been updated to compile with a C++ compiler, but uses none of the new features of the new language.)

  This package is useful for some special operations.

C++ I/O

C++ provides four standard class variables for standard I/O.

Variable	Use
`std::cin`	Console in (standard input)
`std::cout`	Console output (standard output)
`std::cerr`	Console error (standard error)
`std::clog`	Console log

Normally `std::cin` reads from the keyboard, and `std::cout`, `cerr`, and `clog` go to the screen.

Most operating systems allow I/O redirection

```
my_prog <file.in
my_prog >file.out
```

File I/O

File I/O is accomplished using the `<fstream.h>` package.
Input is done with the `ifstream` class and output with the `ofstream` class.

Example:

```
ifstream data_file;    // File for reading the data from
data_file.open("numbers.dat");
for (i = 0; i < 100; ++i)
    data_file >> data_array[i];
data_file.close();
```

Open can be consolidated with the constructor.

```
ifstream data_file("numbers.dat");
```

The close will automatically be done by the destructor.
To check for open errors:

```
if (data.file.bad()) {
    cerr << "Unable to open numbers.dat\n";
    exit (8);
}
```

Reading Number

```
/*********************************************************

 *********************************************************/

int main(){

    }

}
```

`getline` member function

```
istream &getline(char *buffer, int len, char delim = '\n')
```

Parameters:

`buffer` A buffer to store the data that has been read.

`len` Length of the buffer in bytes. The function reads up to len-1 bytes of data into the buffer. (One byte is reserved for the terminating null character: '\0') This parameter is usually `sizeof(buffer)`.

`delim` The character used to signal end of line.

The `sizeof` operator returns the size (in bytes) of a variable or type.

Output files

Example:

```
ofstream out_file("out.dat");
```

Full constructor:

```
ofstream::ofstream(const char *name, int mode=ios::out,
                   int prot = filebuf::openprot);
```

Parameters:

`name` The name of the file.

`mode` A set of flags or'ed together that determine the open mode. The flag `ios::out` is required for output files.

`prot` File protection. This is an operating system dependent value that determines the protection mode for the file. On UNIX the protection defaults to 0644 (read/write owner, group read, others read). For MS-DOS/Windows this defaults to 0 (Normal file).

Open Flag

Flag	Meaning
`std::ios::app`	Append data to the end of the output file.
`std::ios::ate`	Goto the end of file when opened.
`std::ios::in`	Open for input (must be supplied to opens for ifstream variables)
`std::ios::out`	Open file for output (must be supplied to ofstream opens).
`std::ios::binary`	Binary file (if not present, the file is opened as an ASCII file).
`std::ios::trunc`	Discard contents of existing file when opening for write.
`std::ios::nocreate`	Fail if the file does not exist (output files only. Input files always fail if there is no file.)
`std::ios::noreplace`	Do not overwrite existing file. If a file exists, cause the open to fail.

Example:

```
ofsteam out_file("data.new",
      ios::out|ios::binary|ios::nocreate|ios::app);
```

Conversion Routines

To print a number such as 567 you must turn it into three characters “5”, “6” and “7”.

The << operator converts numbers to characters and writes them. Conversion is controlled by a number of flags set by the `setf` and `unsetf` member function calls.

```
file_var.setf(flags);   // Set flags
file_var.unsetf(flags); // Clear flags
```

Conversion Flags

Flag	Meaning
`std::ios::skipws`	Skip leading whitespace characters on input.
`std::ios::left`	Output is left justified
`std::ios::right`	Output is right justified
`std::ios::internal`	Numeric output is padded by inserting a fill character between the sign or base character and the number itself.
`std::ios::dec`	Output numbers in base 10, decimal format
`std::ios::oct`	Output number in base 8, octal format.
`std::ios::hex`	Output numbers in base 16, hexadecimal format.
`std::ios::showbase`	Print out a base indicator at the beginning of each number. For example: hexadecimal numbers are proceeded with a “0x”.
`std::ios::showpoint`	Show a decimal point for all floating point numbers whether or not it’s needed
`std::ios::uppercase`	When converting hexadecimal numbers show the digits A-F as upper case.
`std::ios::showpos`	Put a plus sign before all positive numbers.
`std::ios::scientific`	Convert all floating point numbers to scientific notation on output.
`std::ios::fixed`	Convert all floating point numbers to fixed point on output.
`std::ios::unitbuf`	Buffer output. (More on this later)
`std::ios::stdio`	Flush stream after each output.

Example

```
number = 0x3FF;
std::cout << "Dec: " << number << '\n';
std::cout.setf(ios::hex);
std::cout << "Hex " << number << '\n';
std::cout.setf(ios::dec);
```

Output:

```
Dex 1023
Hex 3ff
```

Other Conversion Control Functions

Controlling the width of the output:

```
int file_var.width(int size);
```

Controlling the precision of floating point (number of digits after the point).

```
int file_var.precision(int digits);
```

Setting the fill character:

```
char file_var.fill(char pad);
```

I/O Manipulators

```
#include <iostream>
#include <iomanip.h>

number = 0x3FF;
std::cout << "Number is " << hex <<
                 number << dec << '\n';
```

I/O Manipulators

Manipulator	Description
`std::setiosflags(long flags)`	Set selected conversion flags
`std::resetiosflags(long flags)`	Reset selected flags.
`std::dec`	Output numbers in decimal format.
`std::hex`	Output numbers in hexadecimal format.
`std::oct`	Output numbers in octal format.
`std::setbase(int base)`	Set conversion base to 8, 10, or 16. Sort of a generalized `dec`, `hex`, `oct`.
`std::setw(int width)`	Set the width of the output.
`std::setprecision(int precision)`	Set the precision of floating point output
`std::setfill(char ch)`	Set the fill character
`std::ws`	Skip whitespace on input
`std::endl`	Output end of line (‘\n’)
`std::ends`	Output end of string (‘\0’)
`std::flush`	Force any buffered output out.

I/O Example

```
int main()
{









}
```

Output:

```
123456789012345678901234567890
12<-
   12<-
***12<-
+12**<-
12.34<-
12.3<-
1e+01<-
```

Binary and ASCII files

ASCII Files

- Contain characters you can read
- Can be printed directly on the printer
- Take up lots of space
- Portable

Binary files

- Contain the “raw” data
- You can’t read them, they print garbage on the screen if you try to type them.
- Can not be printed directly on a printer.
- Relatively compact.
- Mostly machine dependent.

Binary vs. Character

In C++ we use the notation: ‘1’ to represent the character one. We represent the number as: 1.

The character ‘1’ has the numeric value 49.

To turn characters into numbers, we need to subtract 48 or the value of the

```
int integer;
char ch;

ch = '5';
integer = ch - 48;
std::cout << "Integer " << integer << '\n';
```

’0’ is 48, you can just subtract ’0’.

End of Line Puzzle

In the dark ages, BC (Before Computers) teletypes used <carriage return>

When computers came into existence storage cost $$$ so some people decided to cut the end of line to one character.

UNIX	Uses <line feed> only for end of line
Apple	Uses <carriage return> only for end of line
MS/DOS	Uses <carriage return><line feed> for end of line.

When reading ASCII files, the end-of-line must be translated into a new line character ‘\n‘. Binary files do not need this translation. Translation of binary files causes problems.

Binary vs. ASCII opens

```
// open ASCII file for reading
ascii_file.open("name", ios::in);

// open binary file for reading
binary_file.open("name", ios::in|ios::binary);
```

We write 128 bytes. DOS gets 129. Why?

```
int main()
{









    }


    }

}
```

Hint: Here is a hex dump of the MS-DOS/Windows file:

Dump of MS-DOS output

```
080:7f
```

Binary Input

```
in_file.read(data_ptr  size);
```

data_ptr Pointer to a place to put the data.
size Number of bytes to be read.
Example:

```
if (in_file.bad()) {


}
if (in_file.gcount() != sizeof(rectangle)) {
    cerr << "Error: Unable to read full rectangle\n";
    cerr << "I/O error of EOF encounterd\n ";
}
```

Binary Output is similar:

```
out_file.write(data_ptr  size);
```

Buffering Problems

When will the output be printed?

```
std::cout << "Starting";
do_step_1();

do_step_2();

do_step_3();
```

Print it now:

```
std::cout << "Starting" << std::flush;
do_step_1();

do_step_2();

do_step_3();
```

Unbuffered I/O

How to pick up a bunch of paper clips. (Buffered input)

1. Pick up a paper clip in your left hand.
2. Put in your right hand.
3. Repeat the last two steps until the right hand (buffer) is full.
4. Dump the handful in the box.

How to pick up cannon balls (unbuffered I/O)

1. Pick up cannon ball using both hands.
2. Dump it in the box. Be careful to avoid dropping it on your feet.

Buffered I/O is useful for small things. Unbuffered works for larger reads and writes.

Unbuffered I/O routines

```
name  flags
name  flags  mode
```

file_descriptor

An integer that is used to identify the file for the read, write and close calls. If file descriptor is less than 0 an error occurred.

name Name of the file.

flags Defined in the `fcntl.h` header file.

mode Protection mode for the file. Normally this is 0666 for most files.

Open Flags

Flag	Meaning
O_RDONLY	Open for reading only.
O_WRONL Y	Open for writing only.
O_RDWR	Open for reading and writing.
O_APPEND	Append new data at the end of the file.
O_CREAT	Create file (*mode* file required when this flag present).
O_TRUNC	If the file exists, truncate it to 0 length.
O_EXCL	Fail if file exists.
O_BINARY	Open in binary mode (Older UNIX systems may not have this flag).

Open examples:

```
data_fd = open("data.txt", O_RDONLY);
 out_fd = open("output.dat", O_CREAT|O_WRONLY, 0666);
```

Pre-opened files:

File Number	Description
0	Standard in
1	Standard out
2	Standard error

Read function

```
read_size = read(file_descriptor, buffer, size);
```

read_size The actual number of bytes read. A 0 indicates end of file and a negative number indicates an error.

file_descriptor

File descriptor of an open file.

buffer Pointer to the place to read the data.

size Size of the data to be read. This is the size of the request. The actual number of bytes read may be less that this. (For example, we may run out of data.)

Write and close functions

```
write_size = write(file_descriptor, buffer,
size);
```

write_size

Actual number of bytes written. A negative number indicates an error.

file_descriptor

File descriptor of an open file.

buffer Pointer to the data to be written.

```
flag = close(file_descriptor)
```

flag 0 for success, negative for error.

file_descriptor

file_descriptor of an open file.

Copy program

```
/*****************************************

*****************************************/
```

Copy Program

```
            exit(8);
        }


            exit(8);
        }


            exit(8);
        }









                exit(8);
            }

        }
        close(in_file);
        close(out_file);

    }
```

Designing file formats

We need a configuration file for a graph program.
One layout:

height (in inches)
width (in inches)
x lower limit
x upper limit
y lower limit
y upper limit
x scale
y scale

Sample file:

```
10.0
7.0
0
100
30
300
0.5
2.0
```

C Style I/O Routines

File variables:

The declaration for a file variable is:

```
#include <stdio.h>
FILE *file-variable;      /* comment */
```

Open function:

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

file-variable

A file variable.

name Actual name of the file (*data.txt*, *temp.dat*, etc.).

mode Indicates if the file is to be read or written. Mode is “w” for writing and “r” for reading.

Close function:

```
status = fclose(file-variable);
```

C's standard files

File	Description
stdin	Standard input (open for reading). Equivalent to C++'s std::cin
stdout	Standard output (open for writing). Equivalent to C++'s std::cout
stderr	Standard error (open for writing). Equivalent to C++'s std::cerr
	There is no C file equivalent to C++'s std::clog

Counting Characters

```
int main()
{









        exit(8);
    }



            break;
        ++count;
    }


    fclose(in_file);

}
```

Note: The function `fgetc` gets a single character or returns the *integer* `EOF` if there are none left.

Other functions

Writing a character:

```
fputc(character, file);
```

Getting a string:

```
string_ptr = fgets(string, size, file);
```

string_ptr Equal to `string` if the read was successful, or NULL if EOF or an error is detected.

string A character array where the function places the string.

size The size of the character array. Fgets reads until it gets a `size-1` characters. It then ends the string with a null ('\0') .

Writing a string:

```
string_ptr = fputs(string, file);
```

C Conversion Routines

Printing

```
printf(format, parameter-1, parameter-2, ...);
```

Example:

```
printf("Hello World\n");
```

prints:

```
Hello World
```

Example:

```
printf(“The answer is %d\n”, answer);
```

Conversion Characters

Conversion	Variable Type
`%d`	`int`
`%ld`	`long int`
`%d`	`short int`
`%f`	`float`
`%lf`	`double`
`%u`	`unsigned int`
`%lu`	`unsigned long int`
`%u`	`unsigned short int`
`%s`	`char *` (string)
`%c`	`char`
`%o`	`int` (prints octal)
`%x`	`int` (prints in hexadecimal)
`%e`	`float` (in the form *d.ddd*E+*dd*)

Why does 2+2=5986?

```
int main() {




}
```

Why does 21/7 = 0

```
int main(){




}
```

Printing to a file

```
fprintf(file, format, parameter-1,
                      parameter-2, ...);
```

"Printing" to a string:

```
sprintf(string, format, parameter-1,
                  parameter-2, ...);
```

Example:

```
char string[40];          /* the file name */

/* current file number for this segment */
int file_number = 0;

sprintf(string, "file.%d", file_number);
++file_number;
out_file = fopen(string, "w");
```

Reading data

```
number = fscanf(file, format, &parameter-1, . . .);
```

number Number of parameters successfully converted.
file A file opened for reading.
format Describes the data to be read.
parameter-1
First parameter to be read.

WARNING:

If you forget to put & in front of each variable for `scanf`, the result can be a "Segmentation violation core dumped" or "Illegal memory access" error. In some cases a random variable or instruction will be modified. This is not common on UNIX machines, but MS-DOS/Windows, with its lack of memory protection, cannot easily detect this problem. On MS-DOS/Windows, omitting & can cause a system crash.

Don't use `fscanf`

The end of line handling in `fscanf` is so weird that it's almost impossible to get the end of line right. To avoid the problems with `fscanf`, don't use it.

Instead use `fgets` / `sscanf`.

```
char line[100];    // Line for data

// Read numbers
fgets(line, sizeof(line), stdin);
sscanf(line, "%d %d", &number1, &number2);
```

C Style Binary I/O

```
read_size = fread(data_ptr, 1, size, file);
```

read_size	Size of the data that was read. If this is less than size, then an end of file or error occurred.
data_ptr	Pointer to the data to be read.
size	Number of bytes to be read.
file	Input file.

```
}
```

Writing:

```
write_size = fwrite(data_ptr, 1, size, file);
```