**C++ File I/O**

A file is an organized collection of related data stored on secondary storage device. Data files are used to handle voluminous data.

<fstream> and the File Classes

To perform file I/O, you must include the header **<fstream>** in your program. It

defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios.**

Opening and Closing a File

First method—using open()

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream. There are three types of streams: input, output, and input/output.

To create an input stream, you must declare the stream to be of class **ifstream**. To

create an output stream, you must declare it as class **ofstream**. Streams that will be

performing both input and output operations must be declared as class **fstream**.

For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

**ifstream in; // input**

**ofstream out; // output**

**fstream io; // input and output**

Once you have created a stream, one way to associate it with a file is by using **open()** as shown below. This function is a member of each of the three stream classes.

**Streamobject.open(*filename*, open*mode*);**

Here, *filename* is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values defined by **openmode**, which is an enumeration defined by **ios** (through its base class **ios\_base**).

ios::app

ios::ate

ios::binary

ios::in

ios::out

ios::trunc

You can combine two or more of these values by ORing them together.

Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output.

Including **ios::ate** causes a seek to the end of the file to occur when the file is opened. Although **ios::ate** causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file.

The **ios::in** value specifies that the file is capable of input.

The **ios::out** value specifies that the file is capable of output.

The **ios::binary** value causes a file to be opened in binary mode.

By default, all files are opened in text mode. In text mode, various character translations may take place, such as carriage return/linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur. Understand that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length. When creating an output stream using **ofstream**, any preexisting file by that name is automatically truncated.

The following fragment opens a normal output file.

**ofstream out;**

**out.open("test", ios::out);**

For **ifstream**, *mode* defaults to **ios::in**; for **ofstream**, it is **ios::out | ios::trunc**; and for **fstream**, it is**ios::in | ios::out**. Therefore, the preceding statement will usually look like this:

**out.open("test"); //** defaults to output and normal file

*Depending on your compiler, the mode parameter for* ***fstream::open( )*** *may not*

*default to* ***in | out****. Therefore, you might need to specify this explicitly.*

**Second Method—using constructor of steam class**

Although it is entirely proper to open a file by using the **open()** function, most of the time you will not do so because the **ifstream**, **ofstream**, and **fstream** classes have constructor functions that automatically open the file. The constructor functions have the same parameters and defaults as the **open()** function. Therefore, you will most commonly see a file opened as shown here:

**ifstream mystream("myfile");** // open file for input

Note

**Checking for File Open Error**

If **open()** fails, the stream will evaluate to false when used in a Boolean expression.

Therefore, before using a file, you should test to make sure that the open operation

succeeded. You can do so by using a statement like this:

**if(!mystream) {**

**cout << "Cannot open file.\n";**

**// handle error**

**}**

You can also check to see if you have successfully opened a file by using the **is\_open()** function, which is a member of **fstream**, **ifstream**, and **ofstream**. It has this prototype:

**bool is\_open( );**

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if **mystream** is currently open:

**if(!mystream.is\_open()) {**

**cout << "File is not open.\n";**

**// ...**

**Closing a File**

To close a file, use the member function **close()** . For example, to close the file linked to a stream called **mystream**, use this statement:

**mystream.close();**

The **close()** function takes no parameters and returns no value.

Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the **<<** and **>>** operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file. For example, this program creates a short inventory file that contains each item's name and its cost:

**Reading and Writing Text Files**

**#include <iostream>**

**#include <fstream>**

**using namespace std;**

**int main()**

**{**

**ofstream out("INVNTRY"); // output, normal file**

**if(!out) {**

**cout << "Cannot open INVENTORY file.\n";**

**return 1;**

**}**

**out << "Radios " << 39.95 << endl;**

**out << "Toasters " << 19.95 << endl;**

**out << "Mixers " << 24.80 << endl;**

**out.close();**

**ifstream in("INVNTRY"); // input**

**if(!in) {**

**cout << "Cannot open INVENTORY file.\n";**

**return 1;**

**}**

**char item[20];**

**float cost;**

**in >> item >> cost;**

**cout << item << " " << cost << "\n";**

**in >> item >> cost;**

**cout << item << " " << cost << "\n";**

**in >> item >> cost;**

**cout << item << " " << cost << "\n";**

**in.close();**

**return 0;**

**}**

When reading text files using the **>>** operator, keep in mind that certain character

translations will occur. For example, white-space characters are omitted. If you want to prevent any character translations, you must open a file for binary access and use the functions discussed in the next section.

When inputting, if end-of-file is encountered, the stream linked to that file will

evaluate as false.

**Unformatted and Binary I/O**

**put( ) and get( )**

**istream &get(char &*ch*);**

**ostream &put(char *ch*);**

The **get()** function reads a single character from the invoking stream and puts that

value in *ch*. It returns a reference to the stream. The **put()** function writes *ch* to the

stream and returns a reference to the stream.

**// using put() to write all characters from zero to 255 to a file called CHARS.**

**#include <iostream>**

**#include <fstream>**

**using namespace std;**

**int main()**

**{**

**int i;**

**ofstream out("chars", ios::out | ios::binary);**

**if(!out) {**

**cout << "Cannot open output file.\n";**

**return ;**

**}**

**// write all characters to disk**

**for(i=0; i<256; i++) out.put((char) i);**

**out.close();**

**// display the contents of “chars” using get()**

**char ch;**

**ifstream in(“datafile”, ios::in | ios::binary);**

**if(!in) {**

**cout << "Cannot open file.";**

**return 1;**

**}**

**while(in) { // in will be false when eof is reached**

**in.get(ch);**

**if(in) cout << ch;**

**}**

**return ;**

**}**

**read( ) and write( )**

Another way to read and write **blocks of binary data** is to use C++'s **read()** and **write()** functions. Their prototypes are

**istream &read(char *\*buf*, streamsize *num*);**

**ostream &write(char *\*buf*, streamsize *num*);**

The read() function reads num characters from the invoking stream and puts them in the buffer pointed to by buf. The write() function writes num characters to the invoking stream from the buffer pointed to by buf. As mentioned in the preceding chapter, streamsize is a type defined by the C++ library as some form of integer. It is capable of holding the largest number of characters that can be transferred in any one I/O operation.

//Writes a structure to disk and then reads it back in:

**#include <iostream>**

**#include <fstream>**

**#include <cstring>**

**using namespace std;**

**struct status {**

**char name[80];**

**double balance;**

**unsigned long account\_num;**

**};**

**int main()**

**{**

**struct status acc;**

**strcpy(acc.name, "NMIT");**

**acc.balance = 1123.23;**

**acc.account\_num = 34235678;**

**// write data**

**ofstream outbal("balance", ios::out | ios::binary);**

**if(!outbal) {**

**cout << "Cannot open file.\n";**

**return 1;**

**}**

**outbal.write((char \*) &acc, sizeof(struct status));**

**outbal.close();**

**// now, read back;**

**ifstream inbal("balance", ios::in | ios::binary);**

**if(!inbal) {**

**cout << "Cannot open file.\n";**

**return 1;**

**}**

**inbal.read((char \*) &acc, sizeof(struct status));**

**cout << acc.name << endl;**

**cout << "Account # " << acc.account\_num;**

**cout << endl << "Balance: " << acc.balance;**

**inbal.close();**

**return 0;**

**}**

Exercise: modify above program to write and read an array of structure usin block IO functions

**Detecting EOF**

You can detect when the end of the file is reached by using the member function **eof(),** which has this prototype:

**bool eof( );**

It returns true when the end of the file has been reached; otherwise it returns false.

The following program uses **eof()** to display the contents of a file in both hexadecimal and ASCII.

**if(in.eof()) {**

**cout << "END OF FILE\n";**

**return ;**

**}**

**The ignore( ) Function**

You can use the **ignore()** member function to read and discard characters from the

input stream. It has this prototype:

**istream &ignore(streamsize *num*=1, int\_type *delim*=EOF);**

It reads and discards characters until either *num* characters have been ignored (1 by

default) or the character specified by *delim* is encountered (**EOF** by default). If the

delimiting character is encountered, it is not removed from the input stream. Here,

**int\_type** is defined as some form of integer.

//The next program reads a file called TEST. It ignores characters until either a space is encountered or 10 characters have been read. It then displays the rest of the file.

**#include <iostream>**

**#include <fstream>**

**using namespace std;**

**int main()**

**{**

**ifstream in("test");**

**if(!in) {**

**cout << "Cannot open file.\n";**

**return 1;**

**}**

**/\* Ignore up to 10 characters or until first**

**space is found. \*/**

**in.ignore(10, ' ');**

**char c;**

**while(in) {**

**in.get(c);**

**if(in) cout << c;**

**}**

**in.close();**

**return 0;**

**}**

**peek( ) and putback**( )

You can obtain the next character in the input stream without removing it from that

stream by using **peek().** It has this prototype:

**int\_type peek( );**

It returns the next character in the stream or **EOF** if the end of the file is encountered.(**int\_type** is defined as some form of integer.)

Ex:

ifstream in("datafile”);

char ch=in.peek();

You can return the last character read from a stream to that stream by using

**putback().** Its prototype is

**istream &putback(char *c*);** where *c* is the last character read.

Ex:putback(ch);

**flush( )**

When output is performed, data is not necessarily immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling flush() . Its prototype is

**ostream &flush( );**

ex: ofstream out("datafile”);

out.flush();

**In C++'s I/O system, you perform random access by using the seekg() and seekp() functions.** Their most common forms are

**istream &seekg(off\_type *offset*, seekdir *origin*);**

**ostream &seekp(off\_type *offset*, seekdir *origin*);**

The **seekg()** function moves the associated file's current get pointer *offset* number of characters from the specified *origin*, which must be one of these three values:

**ios::beg Beginning-of-file**

**ios::cur Current location**

**ios::end End-of-file**

The **seekp()** function moves the associated file's current put pointer *offset* number

of characters from the specified *origin*.

// program uses both **seekp()** and **seekg()** to reverse the first *<num>*

characters in a file.

**#include <iostream>**

**#include <fstream>**

**#include <cstdlib>**

**using namespace std;**

**int main()**

**{**

**fstream inout(“file”, ios::in | ios::out | ios::binary);**

**if(!inout) {**

**cout << "Cannot open input file.\n";**

**return 1;**

**}**

**long N, i, j;**

**char c1, c2;**

**cin>> N**

**for(i=0, j=N-1; i<j; i++, j--) {**

**inout.seekg(i, ios::beg);**

**inout.get(c1);**

**inout.seekg(j, ios::beg);**

**inout.get(c2);**

**inout.seekp(i, ios::beg);**

**inout.put(c2);**

**inout.seekp(j, ios::beg);**

**inout.put(c1);**

**}**

**inout.close();**

**return 0;**

**}**

To use the program, specify the name of the file that you want to reverse, followed

by the number of characters to reverse. For example, to reverse the first 10 characters of a file called TEST, use this command line:

If the file had contained this:

**This is a test.**

it will contain the following after the program executes if N is **10**

:

**a si sihTtest.**

Obtaining the Current File Position

You can determine the current position of each file pointer by using these functions:

**pos\_type tellg( );**

**pos\_type tellp( );**

**I/O Status**

The C++ I/O system maintains status information about the outcome of each I/O

operation. The current state of the I/O system is held in an object of type **iostate**, which is an enumeration defined by **ios** that includes the following members.



There are two ways in which you can obtain I/O status information. First, you can

call the **rdstate()** function. It has this prototype:

**iostate rdstate( );**

It returns the current status of the error flags. As you can probably guess from looking at the preceding list of flags, **rdstate()** returns **goodbit** when no error has occurred. Otherwise, an error flag is turned on.

The following program illustrates **rdstate()** . It displays the contents of a text file. If an error occurs, the program reports it, using **checkstatus()** .

**#include <iostream>**

**#include <fstream>**

**using namespace std;**

**void checkstatus(ifstream &in);**

**int main()**

**{**

**ifstream in(“datafile”);**

**if(!in) {**

**cout << "Cannot open input file.\n";**

**return 1;**

**}**

**char c;**

**while(in.get(c)) {**

**if(in) cout << c;**

**checkstatus(in);**

**}**

**checkstatus(in); // check final status**

**in.close();**

**return 0;**

**}**

**void checkstatus(ifstream &in)**

**{**

**ios::iostate i;**

**i = in.rdstate();**

**if(i & ios::eofbit)**

**cout << "EOF encountered\**

**else if(i & ios::failbit)**

**cout << "Non-Fatal I/O error\n";**

**else if(i & ios::badbit)**

**cout << "Fatal I/O error\n";**

**}**

This program will always report one "error." After the **while** loop ends, the final

call to **checkstatus()** reports, as expected, that an **EOF** has been encountered. You

might find the **checkstatus()** function useful in programs that you write.

The other way that you can determine if an error has occurred is by using one or

more of these functions:

**bool bad( );**

**bool eof( );**

**bool fail( );**

**bool good( );**

The **bad()** function returns true if **badbit** is set. The **eof()** function was discussed

earlier. The **fail()** returns true if **failbit** is set. The **good()** function returns true if there are no errors. Otherwise, it returns false. Once an error has occurred, it may need to be cleared before your program continues. To do this, use the **clear()** function, which has this prototype:

**void clear(iostate *flags*=ios::goodbit);**

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* as you desire.

**Customized I/O and Files—creating a phonebook and displaying**

#include <iostream>

#include <fstream>

#include <cstring>

using namespace std;

class phonebook {

char name[80];

char areacode[4];

char prefix[4];

char num[5];

public:

phonebook() { };

phonebook(char \*n, char \*a, char \*p, char \*nm)

{

strcpy(name, n);

strcpy(areacode, a);

strcpy(prefix, p);

strcpy(num, nm);

}

friend ostream &operator<<(ostream &stream, phonebook o);

friend istream &operator>>(istream &stream, phonebook &o);

};

// Display name and phone number.

ostream &operator<<(ostream &stream, phonebook o)

{

stream << o.name << " ";

stream << "(" << o.areacode << ") ";

stream << o.prefix << "-";

stream << o.num << "\n";

return stream; // must return stream

}

// Input name and telephone number.

istream &operator>>(istream &stream, phonebook &o)

{

cout << "Enter name: ";

stream >> o.name;

cout << "Enter area code: ";

stream >> o.areacode;

cout << "Enter prefix: ";

stream >> o.prefix;

cout << "Enter number: ";

stream >> o.num;

cout << "\n";

return stream;

}

int main()

{

phonebook a;

char c;

fstream pb("phone", ios::in | ios::out | ios::app);

if(!pb) {

cout << "Cannot open phone book file.\n";

return 1;

}

for(;;) {

do {

cout << "1. Enter numbers\n";

cout << "2. Display numbers\n";

cout << "3. Quit\n";

cout << "\nEnter a choice: ";

cin >> c;

} while(c<'1' || c>'3');

switch(c) {

case '1':

cin >> a;

cout << "Entry is: ";

cout << a; // show on screen

pb << a; // write to disk

break;

case '2':

char ch;

pb.seekg(0, ios::beg);

while(!pb.eof()) {

pb.get(ch);

if(!pb.eof()) cout << ch;

}

pb.clear(); // reset eof

cout << endl;

break;

case '3':

pb.close();

return 0;

}

}

}