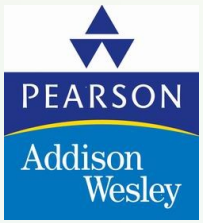
Starting Out with C++: Early Objects
5th Edition

Chapter 13
Advanced File and I/O Operations



Topics

13.1 Files

13.2 Output Formatting

13.3 Passing File Stream Objects to Functions

13.4 More Detailed Error Testing

13.5 Member Functions for Reading and Writing Files

Topics (continued)

13.6 Working with Multiple Files

13.7 Binary Files

13.8 Creating Records with Structures

13.9 Random-Access Files

13.10 Opening a File for Both Input and Output

Steps to Using a File

1. Open the file
2. Use (read from, write to) the file
3. Close the file

File Stream Objects

- Use of files requires file stream objects
- There are three types of file stream objects
  (1) `ifstream` objects: used for input
  (2) `ofstream` objects: used for output
  (3) `fstream` objects: used for both input and output

File Names

- File name can be a full pathname to file:

```
c:\data\student.dat
```

  tells compiler exactly where to look
- File name can also be simple name:

```
student.dat
```

  this must be in the same directory as the program executable, or in the compiler's default directory

Opening a File

- A file is known to the system by its name
- To use a file, a program needs to connect a suitable stream object to the file. This is known as opening the file
- Opening a file is achieved through the `open` member function of a file stream object

Opening a File for Input

- Create an **`ifstream`** object in your program

```
ifstream inFile;
```

- Open the file by passing its name to the stream's open member function

```
inFile.open("myfile.dat");
```

Getting File Names from Users

- Define file stream object, variable to hold file name

```
ifstream inFile;
char FileName(81);
```

- Prompt user to enter filename and read the filename

```
cout << "Enter filename: ";
cin.getline(FileName, 81);
```

- Open the file

```
inFile.open(FileName);
```

Opening a File for Output

- Create an **`ofstream`** object in your program

```
ofstream outFile;
```

- Open the file by passing its name to the stream's open member function

```
outFile.open("myfile.dat");
```

The `fstream` Object

- **`fstream`** object can be used for either input or output

```
fstream file;
```

- To use **`fstream`** for input, specify **`ios::in`** as the second argument to open

```
file.open("myfile.dat",ios::in);
```

- To use **`fstream`** for output, specify **`ios::out`** as the second argument to open

```
file.open("myfile.dat",ios::out);
```

Opening a File for Input and Output

- **`fstream`** object can be used for both input and output at the same time
- Create the fstream object and specify both **`ios::in`** and **`ios::out`** as the second argument to the open member function

```
fstream file;
file.open("myfile.dat",
          ios::in|ios::out);
```

Opening Files with Constructors

- Stream constructors have overloaded versions that take the same parameters as **`open`**
- These constructors open the file, eliminating the need for a separate call to **`open`**

```
fstream inFile("myfile.dat",
               ios::in);
```

File Open Modes

- File open modes specify how a file is opened and what can be done with the file once it is open
- **`ios::in`** and **`ios::out`** are examples of file open modes, also called file mode flags
- File modes can be combined and passed as second argument of open member function

File Mode Flags

`ios::app`	create new file, or append to end of existing file
`ios::ate`	go to end of existing file; write anywhere
`ios::binary`	read/write in binary mode (not text mode)
`ios::in`	open for input
`ios::out`	open for output

File Open Modes

- Not all combinations of file open modes make sense
- **`ifstream`** and **`ofstream`** have default file open modes defined for them, hence the second parameter to their **`open`** member function is optional

Default File Open Modes

- **`ofstream`:**
  - open for output only
  - file cannot be read from
  - file created if no file exists
  - file contents erased if file exists

- **`ifstream`:**
  - open for input only
  - file cannot be written to
  - open fails if file does not exist

Detecting File Open Errors

- Two methods for detecting if a file open failed

  (1) Call **`fail()`** on the stream

```
inFile.open("myfile");
if (inFile.fail())
  { cout << "Can't open file";
    exit(1);
  }
```

Detecting File Open Errors

(2) Test the status of the stream using the ! operator

```
inFile.open("myfile");
if (!inFile)
  { cout << "Can't open file";
    exit(1);
  }
```

Using `fail()` to detect eof

- Example of reading all integers in a file

```
//attempt a read
int x;  infile >> x;
while (!infile.fail())
{  //success, so not eof
   cout << x;
   //read again
   infile >> x;
}
```

Using >> to detect eof

- To detect end of file, `fail()` must be called immediately after the call to `>>`
- The extraction operator returns the same value that will be returned by the next call to fail:
  - (`infile >> x`) is nonzero if `>>` succeeds
  - (`infile >> x`) is zero if `>>` fails

Detecting End of File

- Reading all integers in a file

```
int x;
while (infile >> x)
{
   // read was successful
   cout >> x;
   // go to top of loop and
   // attempt another read
}
```

13.2 Output Formatting

- Can format with I/O manipulators: they work with file objects just like they work with **`cout`**
- Can format with formatting member functions
- The **`ostringstream`** class allows in-memory formatting into a string object before writing to a file

I/O Manipulators

`left, right`	left or right justify output
`oct, dec, hex`	display output in octal, decimal, or hexadecimal
`endl, flush`	write newline (**`endl`** only) and flush output
`showpos, noshowpos`	do, do not show leading **+** with non-negative numbers
`showpoint, noshowpoint`	do, do not show decimal point and trailing zeroes

More I/O Manipulators

`fixed`, `scientific`	use fixed or scientific notation for floating-point numbers
`setw(n)`	sets minimum field output width to **`n`**
`setprecision(n)`	sets floating-point precision to **`n`**
`setfill(ch)`	uses **`ch`** as fill character

Formatting with Member Functions

- Can also use stream object member functions to format output:

```
gradeFile.width(3); // like
                    // setw(3)
```

- Names of member functions may differ from manipulators.

Formatting with Member Functions

Member Function	Manipulator or Meaning
`width(n)`	`setw(n)`
`precision(n)`	`setprecision(n)`
`setf()`	set format flags
`unsetf()`	disable format flags

`sstream` Formatting

1) To format output into an in-memory string object, include the **`sstream`** header file and create an **`ostringstream`** object

```
#include <sstream>
ostringstream outStr;
```

`sstream` Formatting

2) Write to the **`ostringstream`** object using I/O manipulators, all other stream member functions:

```
outStr << showpoint << fixed
       << setprecision(2)
       << 'S'<< amount;
```

`sstream` Formatting

3) Access the C-string inside the **`ostringstream`** object by calling its `str` member function

```
cout << outStr.str();
```

13.3 Passing File Stream Objects to Functions

- File stream objects keep track of current read or write position in the file

- Always use pass a file object as parameter to a function using pass by reference

Passing File Stream Objects to Functions

```
//print all integers in a file to screen
  void printFile(ifstream &in)
  { int x;
    while(in >> x)
    { out << x << " "; }
  }
```

13.4 More Detailed Error Testing

- Streams have error bits (flags) that are set by every operation to indicate success or failure of the operation, and the status of the stream
- Stream member functions report on the settings of the flags

Error State Bits

- Can examine error state bits to determine file stream status

`ios::eofbit`	set when end of file detected
`ios::failbit`	set when operation failed
`ios::hardfail`	set when an irrecoverable error occurred
`ios::badbit`	set when invalid operation attempted
`ios::goodbit`	set when no other bits are set

Error Bit Reporting Functions

`eof()`	true if **`eofbit`** set, false otherwise
`fail()`	true if **`failbit`** or **`hardfail`** set, false otherwise
`bad()`	true if **`badbit`** set, false otherwise
`good()`	true if **`goodbit`** set, false otherwise
`clear()`	clear all flags (no arguments), or clear a specific flag

13.5 Member Functions for Reading and Writing Files

- Unlike the extraction operator `>>`, these reading functions do not skip whitespace:
  - `getline`: read a line of input
  - `get`: reads a single character
  - `seekg`: goes to beginning of input file

`getline` Member Function

- `getline(char s[ ], int max, char stop ='\n')`
  - `char s[ ]:` Character array to hold input
  - `int max` : Maximum number of characters to read
  - `char stop`: Terminator to stop at if encountered before `max` number of characters is read . Default is `'\n'`

Single Character Input

- `get(char &ch)`

  Read a single character from the input stream. Does not skip whitespace.

```
ifstream inFile;  char ch;
inFile.open("myFile");
inFile.get(ch);
cout << "Got " << ch;
```

Single Character Output

- `put(char ch)`

  Output a character to a file

- Example

```
ofstream outFile;
outFile.open("myfile");
outFile.put('G');
```

Single Character I/O

- To copy an input file to an output file

```
char ch; infile.get(ch);
while (!infile.fail())
{ outfile.put(ch);
  infile.get(ch);
}
infile.close();
outfile.close();
```

Moving About in Input Files

- **`seekg(offset, place)`**
  Move to a given **`offset`** relative to a given **`place`** in the file
  - **`offset`**: number of bytes from **`place`**, specified as a **`long`**
  - **`place`**: location in file from which to compute offset
    - **`ios::beg`**: beginning of file
    - **`ios::end`**: end of the file
    - **`ios::cur`**: current position in file

Rewinding a File

- To move to beginning of file, seek to an offset of zero from beginning of file

```
inFile.seekg(0L, ios::beg);
```

- Error or eof bits will block seeking to the beginning of file. Clear bits first:

```
inFile.clear();
inFile.seekg(0L, ios::beg);
```

13.6 Working with Multiple Files

- Can have more than one file open at a time in a program
- Files may be open for input or output
- Need to define stream object for each file

13.7 Binary Files

- Binary files store data in the same format that a computer has in main memory
- Text files store data in which numeric values have been converted into strings of ASCII characters
- Files are opened in text mode (as text files) by default

Using Binary Files

- Pass the **`ios::binary`** flag to the **`open`** member function to open a file in binary mode

```
infile.open("myfile.dat",ios::binary);
```

- Reading and writing of binary files requires special **`read`** and **`write`** member functions

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

Using `read` and `write`

```
read(char *buffer, int numberBytes)
write(char *buffer, int numberBytes)
```

- `buffer:` holds an array of bytes to transfer between memory and the file
- `numberBytes`: the number of bytes to transfer

Address of the buffer needs to be cast to `char *` using `reinterpret_cast`

Using `write`

- To write an array of 2 doubles to a binary file

```
ofstream
outFile("myfile", ios:binary);
double d[2] = {12.3, 34.5};
outFile.write(
   reinterpret_cast<char *>(d),
   sizeof(d)
   );
```

Using `read`

- To read two 2 doubles from a binary file into an array

```
ifstream inFile("myfile", ios:binary);
 const int DSIZE = 10;
 double data[DSIZE];
 inFile.read(
    reinterpret_cast<char *>(data),
    2*sizeof(double)
    );
 // only data[0] and data[1] contain
 // values
```

13.8 Creating Records with Structures

- Can write structures to, read structures from files
- To work with structures and files,
  - use **`binary`** file flag upon open
  - use **`read`**, **`write`** member functions

Creating Records with Structures

```
struct TestScore
{ int studentId;
  float score;
  char grade;
};
TestScore test1[20];
...
// write out test1 array to a file
gradeFile.write(
  reinterpret_cast<char*>(test1),
  sizeof(test1));
```

13.9 Random-Access Files

- Sequential access: start at beginning of file and go through data in file, in order, to end
  - to access 100th entry in file, go through 99 preceding entries first
- Random access: access data in a file in any order
  - can access 100th entry directly

Random Access Member Functions

- `seekg` (seek get): used with input files
- `seekp` (seek put): used with output files
- Both are used to go to a specific position in a file

Random Access Member Functions

- `seekg(offset,place)`
- `seekp(offset,place)`

  `offset`: long integer specifying number of bytes to move

  `place`: starting point for the move, specified by `ios:beg`, `ios::cur` or `ios:end`

Random-Access Member Functions

- Examples:

```
// Set read position 25 bytes
// after beginning of file
inData.seekg(25L, ios::beg);

// Set write position 10 bytes
// before current position
outData.seekp(-10L, ios::cur);
```

Random Access Information

- **`tellg`** member function: return current byte position in input file

```
int whereAmI;
whereAmI = inFile.tellg();
```

- **`tellp`** member function: return current byte position in output file

```
whereAmI = outFile.tellp();
```

13.10 Opening a File for Both Input and Output

- File can be open for input and output simultaneously
- Supports updating a file:
  - read data from file into memory
  - update data
  - write data back to file
- Use **`fstream`** for file object definition:

```
fstream gradeList("grades.dat",
                  ios::in | ios::out);
```