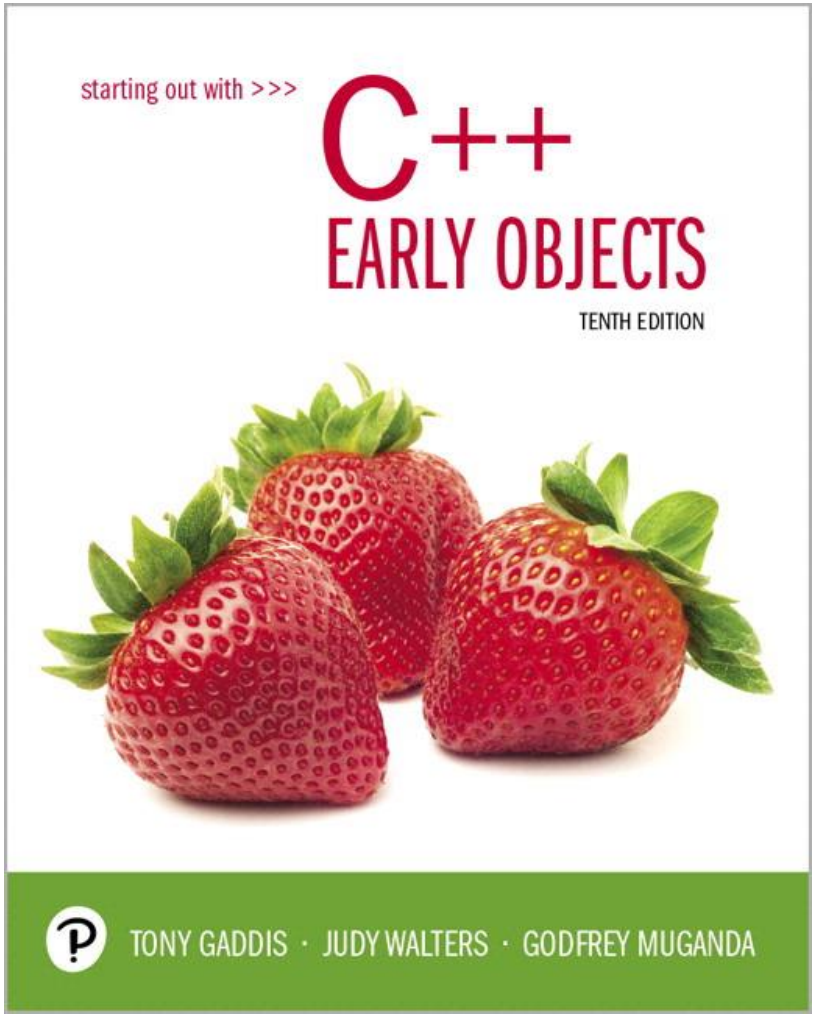
Starting Out with C++ Early Objects

Tenth Edition



Chapter 13

Advanced File and I/O Operations

Topics

13.1 Input and Output Streams

13.2 More Detailed Error Testing

13.3 Member Functions for Reading and Writing Files

13.4 Binary Files

13.5 Creating Records with Structures

13.6 Random-Access Files

13.7 Opening a File for Both Input and Output

13.1 Input and Output Streams

- Input Stream – data stream from which information can be read
  - Ex: **`cin`** and the keyboard
  - Use **`istream`**, **`ifstream`**, and **`istringstream`** objects to read data
- Output Stream – data stream to which information can be written
  - Ex: **`cout`** and computer screen
  - Use **`ostream`**, **`ofstream`**, and **`ostringstream`** objects to write data
- Input/Output Stream – data stream that can be both read from and written to
  - Use **`fstream`** objects here

File Stream Classes

- **`ifstream`** (open primarily for input), **`ofstream`** (open primarily for output), and **`fstream`** (open for either or both input and output)
- All have initialization constructors that take a filename (as a C-string) and an open mode.
- C++ 11: The filename can be a string object.
- Use **`close()`** member functions to disconnect program from an external file when access is no longer needed
  - Files should be open only while being used
  - Always close files before the program ends

File Open Modes

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

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);
```

File Mode Flags

Mode Flag	Meaning
`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. Open fails if file not found
`ios::out`	open for output. Starts with an empty file
`ios::trunc`	open output. Erase/truncate file if exists

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 or to the constructor

```
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

- **`ifstream`**, **`ofstream`**, and **`fstream`** have default file open modes defined for them, hence the second parameter to their **`open`** member function is optional
- Defaults:

  `ifstream` – open for input

  `ofstream` – open for output

  `fstream` – open for both input and output

Output Formatting with I/O Manipulators

- 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 1 of 2

Manipulator	Meaning
`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

I/O Manipulators 2 of 2

Manipulator	Meaning
`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

`ostringstream` Formatting 1 of 3

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 of 3

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

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

`sstream` Formatting 3 of 3

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

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

13.2 More Detailed Error Testing

- Stream objects 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

Examine error state bits to determine file stream status

Bits	When does it occur?
`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

Function	Return value or purpose
`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

Detecting File Operation Errors

- The file stream is set to true if a file operation succeeds. It is set to false when a file operation fails
- You can test the status of the stream by testing the file handle:

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

13.3 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, peek`: read a single character
- `seekg`: go to a location in input file

`getline` Member Function

`getline(source, dest, stopChar)`

- `source:` stream being read from
- `dest` : string to hold the text from source
- `stopChar`: Terminator character. Optional, default is `'\n'`

Single Character Input 1 of 2

`get()`

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

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

Single Character Input 2 of 2

```
get(char &ch)
```

Read a single character from the input stream and put it in **ch**. Does not skip whitespace.

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

Single Character Input, with a Difference

`peek()`

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

```
ifstream inFile;  char ch;
inFile.open("myFile");
ch = inFile.peek();
cout << "Got: " << ch;
ch = inFile.peek();
cout << "Got: " << ch;//same output
```

Single Character Output

- `put(int c)`

  Output a character to a file. The integer code of the character is passed to **`put`**

- Example

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

Example of 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 the 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, then seek:

```
inFile.clear();

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

13.4 Binary Files

- Binary files store data in the same format that a computer uses 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 may need to be cast to `char *` using **`reinterpret_cast <char *>`**

Using `write`

To write an array of 2 doubles to a binary file

```
ofstream oFile("myfile",ios:binary);
double d[2] = {12.3, 34.5};
oFile.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
```

13.5 Creating Records with Structures 1 of 2

- You can write structures to and 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 2 of 2

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

Notes on Structures Written to Files

- Structures to be written to a file must not contain pointers
- Since string objects use pointers and dynamic memory internally, structures to be written to a file must not contain any string objects. Use **`char`** arrays instead.

13.6 Random-Access Files

- Sequential access: start at beginning of file and go through the data the in file, in order, to the end of the file
  - 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 1 of 2

```
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 2 of 2

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, as a **`long`**

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

- **`tellp`** member function: return current byte position in output file, as a **`long`**

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

13.7 Opening a File for Both Input and Output

- A 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);
```

Copyright