Chapter 13: Advanced File and I/O Operations

Starting Out with C++
Early Objects
Global Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Addison-Wesley
is an imprint of
PEARSON

Topics

13.0 Using Files for Data Storage
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

Topics

13.5 Creating Records with Structures

13.6 Random-Access Files

13.7 Opening a File for Both Input and Output

13.8 Online Friendship Connections Case Study: Object Serialization

13.0 Using Files for Data Storage

- We can use a file instead of monitor screen for program output
- Files are stored on secondary storage media, such as disk
- Files allow data to be retained between program executions
- We can later use the file instead of a keyboard for program input

File Types

- Text file – contains information encoded as text, such as letters, digits, and punctuation. Can be viewed with a text editor such as Notepad.
- Binary file – contains binary (0s and 1s) information that has not been encoded as text. It cannot be viewed with a text editor.

File Access – Ways to Use the Data in a File

- Sequential access – read the 1st piece of data, read the 2nd piece of data, …, read the last piece of data. To access the n-th piece of data, you have to retrieve the preceding n pieces first.
- Random (direct) access – retrieve any piece of data directly, without the need to retrieve preceding data items.

What is Needed to Use Files

1. Include the **`fstream`** header file
2. Define a file stream object
   - **`ifstream`** for input from a file

     ```
     ifstream inFile;
     ```
   - **`ofstream`** for output to a file

     ```
     ofstream outFile;
     ```

Open the File

3. Open the file

- Use the **`open`** member function

```
inFile.open("inventory.dat");
outFile.open("report.txt");
```

- Filename may include drive, path info.
- Output file will be created if necessary; existing output file will be erased first
- Input file must exist for **`open`** to work

Use the File

4. Use the file

- Can use output file object and **<<** to send data to a file

```
outFile << "Inventory report";
```

- Can use input file object and **>>** to copy data from file to variables

```
inFile >> partNum;
inFile >> qtyInStock >> qtyOnOrder;
```

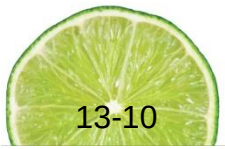
Close the File

5. Close the file

- Use the **`close`** member function

```
inFile.close();
outFile.close();
```

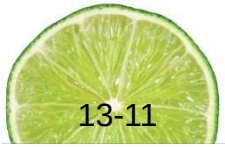
- Don't wait for operating system to close files at program end
  - There may be limit on number of open files
  - There may be buffered output data waiting to be sent to a file that could be lost



Example

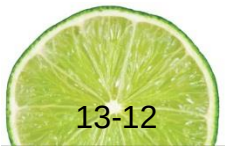
P326 Program 5-17 eP291

P328 Program 5-18, eP293



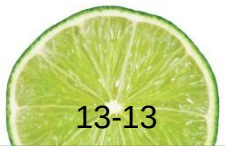
Input File – the Read Position

- Read Position – location of the next piece of data in an input file
- Initially set to the first byte in the file
- Advances for each data item that is read. Successive reads will retrieve successive data items.
- When the >> operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines).



Using Loops to Process Files

- A loop can be used to read data from or write data to a file
- It is not necessary to know how much data is in the file or will be written to the file
- Several methods exist to test for the end of the file



Using the >> Operator to Test for End of File (EOF) on an Input File

- The stream extraction operator (**>>**) returns a true or false value indicating if a read is successful
- This can be tested to find the end of file since the read "fails" when there is no more data
- Example:

```
while (inFile >> score)
   sum += score;
```

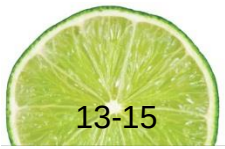
Program 5-21, eP298

File Open Errors

- An error will occur if an attempt to open a file for input fails:
  - File does not exist
  - Filename is misspelled
  - File exists, but is in a different place
- The file stream object is set to true if the open operation succeeded. It can be tested to see if the file can be used:

```
if (inFile)
{
   // process data from file
}  else
   cout << "Error on file open\n";
```

P333 Program 5-21, eP299

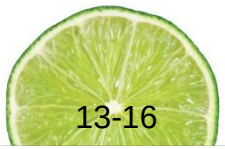


User-Specified Filenames

- Program can prompt user to enter the names of input and/or output files. This makes the program more versatile.
- Filenames can be read into string objects. The C-string representation of the string object can then be passed to the open function:

```
cout << "Which input file? ";
cin >> inputFileName;
inFile.open(inputFileName.c_str());
```

Program 5-23, eP300



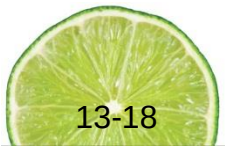
13.1 Input and Output Streams

- 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 monitor 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

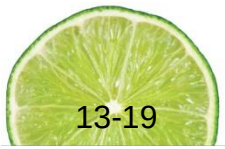
- **`ifstream`** (open primarily for input), **`ofstream`** (open primarily for output), and **`fstream`** (open for either or both input and output)
- All have **`open`** member function to connect the program to an external file
- All have **`close`** member function to disconnect program from an external file when access is finished
  - Files should be open for as short a time as possible
  - Always close files before the program ends

P889 Program 13-1, eP838



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 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

ios::app	Append: Output will always take place at the end of the file.
ios::ate	At end: Output will initially take place at the end of the file.
ios::binary	read/write in binary form
ios::in	open for input
ios::out	open for output

ios : : trunc Truncate: file contents are discarded

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);
```

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

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);
```

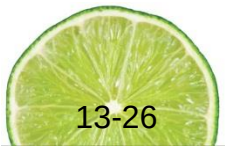
Default File Open Modes

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

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

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

`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

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)
       << '$'<< amount;
```

`sstream` Formatting

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

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

P891 Program 13-2, eP841

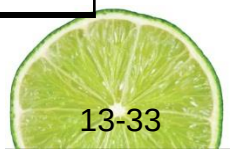
13.2 More Detailed Error Testing

- 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

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

Detecting File Operation Errors

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

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

P897 Program 13-4, eP846

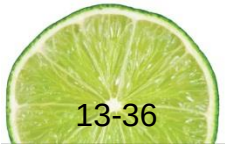
13.3 Member Functions for Reading and Writing Files

Figure 13-1 murphy.txt, eP848

P899 Program 13-5, eP849

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` global Function

```
istream& getline(istream& is,
  string& str, char delim='\n')
```

- `is :` A line of text read from this stream
- `str:` A string variable to hold input
- `char delim`: Terminator to stop at if encountered before maximum number of characters is read . Optional, default is `'\n'`

P901 Program 13-6, eP850

P902 Program 13-7, eP852

Single Character Input

`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, Again

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;
```

P904 Program 13-8, eP853

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
```

P905 Program 13-9, eP854~855

Single Character Output

- `ostream& put(int c)`

  Output a character to a file

- Example

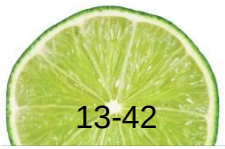
```
ofstream outFile;
outFile.open("myfile");
char ch='G'
outFile.put(ch);
outFile.put(ch+1);
```

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



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();
```

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:

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

P908 Program 13-10, eP858

13.4 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

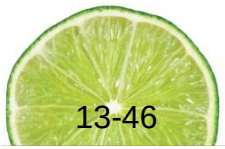
Using Binary Files

- 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 <char *>`

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
```

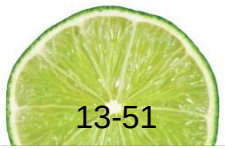
P914 Program 13-11, eP864

13.5 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

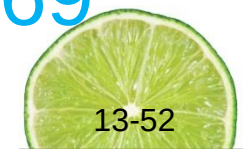
```
struct TestScore
{ int studentId;
  float score;
  char grade;
};
TestScore test1[20];
...
// write out test1 array to a file
gradeFile.write(reinterpret_cast<char*>
  (test1), sizeof(test1));
```



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

P916 Program 13-12 + 13-13, eP866~869



13.6 Random-Access Files

- Sequential access: start at beginning of file and go through the data 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

`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`**

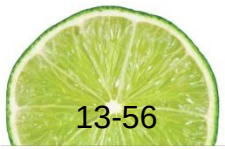
Random-Access Member Functions

- 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);
```

P922 Program 13-14+13-15, eP872~873



Random Access Information

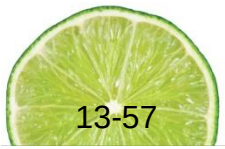
- **`tellg`** member function: return current byte position in input file, as a **`long`**

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

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

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

P926 Program 13-16, eP875



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:

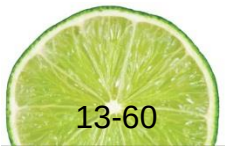
```
fstream gradeList("grades.dat",
                  ios::in | ios::out);
```

13.7 Opening a File for Both Input and Output

- P928 Program 13-17(eP877) create a file with five blank inventory records. (Output)
- P929 Program 13-18(eP878) display the contents of the inventory file on screen. (Input)
- P930 Program 13-19(eP879) edit a specific record in the inventory file. (Input/output)

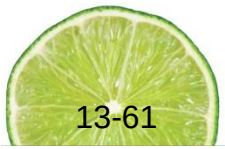
13.7 Online Friendship Connections Case Study: Object Serialization

- **Online Friendship Connections** is an online service that helps people meet and make new friends. People who want to join the club and use its services fill out a registration form, stating their names, age, contact information, gender, hobbies, personal interests, and other pertinent information about themselves. They also specify the qualities they are looking for in a new friend. The service will then try to get two people together if the personal information submitted indicates that there is a high probability of a good match.



13.7 Online Friendship Connections Case Study: Object Serialization

- The process of transforming complex networks of objects interconnected through pointers into a form that can be stored in a disk file (or some other medium outside of central memory) is called object serialization.
- P933 serialization.h, eP882
- P935 serialization.cpp, eP884
- P936 Program 13-20 + Program 13-21, eP885~886



Homework 8

TEXT BOOK CH. 13 Programming Challenges
15. Inventory Program (P947)

Write a program that uses a structure to store the following inventory information in a file:

Item description
Quantity on hand
Wholesale cost
Retail cost
Date added to inventory

The program should have a menu that allows the user to perform the following tasks:

- Add new records to the file.
- Display any record in the file.
- Change any record in the file.