Lecture 20

File Operations

Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

Input/Ouput to and from files

- File input and file output is an essential in programming.
  - Most software involves more than keyboard input and screen user interfaces.
  - Data needs to be stored somewhere when a program is not running, and that means writing data to disk.
  - For this, we need file input and file output techniques.
- Fortunately, this is EASY in C++!
  - If you know how to do screen output with cout, and keyboardinput with cin, then you already know most of it!
  - File I/O with streams works the same way. The primary difference is that objects other than cout and cin will be used

Kinds of Files

- **Formatted Text** vs. **Binary files**
  - A *text* file is simply made of readable text characters.
  - It looks like the output that is typically printed to the screen through the cout object
  - A *binary* file contains unformatted data, saved in its raw memory format. (For example, the integer 123456789 is saved as a 4-byte chunk of data, the same as it's stored in memory - NOT as the 9 digits in this sentence).

Kinds of Files

- **Sequential** vs. **Random Access** files
  - A sequential file is one that is typically written or read fromstart to finish
  - A random access file is one that stores records, all of the same size, and can read or write single records in place, without affecting the rest of the file
- For now, we'll deal with sequential text files

Creating file stream objects, and attaching to files

- `cout` and `cin` are objects
  - `cout` is the standard output stream, usually representing themonitor. It is of type ostream
  - `cin` is the standard input stream, usually representing thekeyboard. It is of type istream and istream are classes
  - If you were to have declared them, you might have written:

```
ostream cout;
istream cin;
```

Creating file stream objects, and attaching to files

- To create file stream objects, we need to include the `<fstream>` library:

```
#include <fstream> using namespace std;
```

- This library has classes `ofstream` ("output file stream") and `ifstream` ("input file stream"). Use these to declare file stream objects:

```
// create file output streams out1 and bob
ofstream out1, bob;
// create file input streams, called in1 and joe
ifstream in1, joe;
```

Creating file stream objects, and attaching to files

- File stream objects need to be attached to files before they can be used. Do this with a member function called open, which takes in the filename as an argument:

```
// For ofstreams, these calls create brand new
// files for output. For ifstreams, these calls
// try to open existings files for input
out1.open("outfile1.txt");
bob.open("clients.dat");
in1.open("infile1.txt");
joe.open("clients.dat");
```

Creating file stream objects, and attaching to files

- Will open() always work?
  - For an input file, what if the file doesn't exist? doesn't have read permission?
  - For an output file, what if the directory is not writable? What if it's an illegal file name?
- Since it's possible for `open()` to fail, one should always check to make sure there's a valid file attached

Creating file stream objects, and attaching to files

- One way is to test the value of the stream object. A stream that is not attached to a valid file will evaluate to "false"

```
//if in1 not attached to a valid source, abort
if (!in1)
{
   cout << "Sorry, bad file.";
   exit(0); // system call to abort program, may
       require <cstdlib> to be included
}
```

Creating file stream objects, and attaching to files

- When finished with a file, it can be detached from the stream object with the member function `close()`:

```
in1.close();
```

- The close function simply closes the file. It does not get rid of the stream object. The stream object can now be used to attach to another file, if desired

Using file streams

- Once a file stream object is attached to a file, it can be used with the same syntax as `cin` and `cout` (for input and output streams, respectively)
- Input file stream usage is like `cin`:

```
int x, y, z;
double a, b, c;
in1 >> x >> y >> z; //read 3 ints from the file
in1 >> a >> b >> c; // read 3 doubles from file
```

Using file streams

- Output file stream usage is like cout:

```
out1 << "Hello, World\n"; // print "Hello, World" to
    the file
out1 << "x + y = " << x + y; // print a math result
    to the file
```

Opening a file in 'append mode'

- The default way for opening an output file is to create a brand new file and begin writing from the beginning
- If another file with the same name already exists, it will be overwritten!
- Existing files can be opened for output, so that the new output is tacked on to the end. This is called appending.

Opening a file in 'append mode'

- To open a file in append mode, we use an extra parameter in the `open()` function:

```
ofstream fout; // create file stream
fout.open("file.txt", ios::app); // open file in
    append mode
```

- There are a number of special constants like this one (`ios::app`). This one will cause a file to be opened for appending

User-entered file names

- File names don't have to be hard-coded as literal strings. We can get file names from other places (like user input, other files, etc), but we need to store them as cstrings.

```
char filename[20];
```

- Filenames are usually in the form of a single word (C++ hates filenames with spaces). So we can just use the extraction operator to read it in.

```
   cin >> filename;
```

User-entered file names

- We can use this variable in the open() function when attaching a file to a stream:

```
ofstream fout;
fout.open(filename);
```

- When error-checking to ensure that a valid file was attached, pick a technique that's appropriate to the situation. If the user just types a filename wrong, we might want to allow them to try again (instead of aborting the program).

Reading Strings

- So far, we have used cin as the input stream for reading strings.
- If we're reading strings from a file, we can use the input filestream instead.
- Assuming the input stream is called in1 and it is attached to a valid input file,

```
//reading in a cstring
char value[100];
in1.getline(value, 100, '\n');

//reading in a string object
string text;
getline(in1, text, '\n');
```

`eof()` member function

- A useful member function of the input stream classes is `eof()`
  - Stands for end of file
  - Returns a bool value, answering the question "Are we at the end of the file?" (or is the "end-of-file" character the next one on the stream?)
  - Can be used to indicate whether the end of an input file has been reached, when reading sequentially

eof() member function

- Very useful when reading files where the size of the file or the amount of data to be read is not known in advance

```
while (!in1.eof()) // while not at end of file
{
   // read and process input from the file
}
```

`eof()` member function

- Can also be used with `cin`, where the user types a key combination representing the "end-of-file" character
  - On Unix and macOS systems, type ctrl-d to enter the end-of-filecharacter
  - On Windows, type ctrl-z to enter the end-of-file character

Character I/O - Output

- We've already used the insertion operator to print characters:

```
char letter = 'A';
cout << letter;
```

- There is also a member function (of output stream classes) called `put()`, which can be used to print a character. It's prototype is:

```
   ostream& put(char c);
```

- Sample calls:

```
char ch1 = 'A', ch2 = 'B', ch3 = 'C';
cout.put(ch1); // equivalent to: cout << ch1;
cout.put(ch2); // equivalent to: cout << ch2;
```

Character I/O - Output

```
ostream& put(char c); // why?
```

- It can be cascaded, like the insertion operator:

```
cout.put(ch1).put(ch2).put(ch3);
```

- The put() function doesn't really do anything more special than the insertion operator does. It's just listed here for completeness

Character I/O - Output

- There are many versions of the extraction operator >>, for reading data from an input stream. This includes a version that reads characters:

```
char letter;
cin >> letter;
```

- However, if we, for example, tried to copy a file into another by reading one character at a time, the output file wouldn't have any whitespace.
- All built-in versions of the extraction operator for input streams will ignore leading white space by default

Character I/O - Output

Here are some other useful member functions (of input stream classes) for working with the input of characters:

- `peek()` - this function returns the ascii value of the next character on the input stream, but does not extract it
- `get()` - the two get functions both extract the next single character on the input stream, and they do not skip any white space.
  - The version with no parameters returns the ascii value of the extracted character
  - The version with the single parameter stores the character in the parameter, passed by reference. Returns a reference to the stream object (or 0, for end-of-file)

Character I/O - Output

- `ignore()` - member function - skips either a designated numberof characters, or skips up to a specified delimiter.
- `putback()` - member function - puts a character back into the input stream

Examples

```
char ch1, ch2, ch3;
cin >> ch1 >> ch2 >> ch3; // reads three characters,
    skipping white space

//get(): no parameters, no white space skipped
ch1 = cin.get();
ch2 = cin.get();
ch3 = cin.get();

//get(): one parameter, can be cascaded
cin.get(ch1).get(ch2).get(ch3);
```

Examples

```
//peek(): trying to read a digit, as a char
char temp = cin.peek(); // look at next character
if (temp < '0' || temp > '9')
   cout << "Not a digit";
else
   ch1 = cin.get(); // read the digit
```

Passing Stream Objects into Functions

- In a function prototype, any type can be used as a formal parameter type or as a return type.
  - This includes classes, which are programmer-defined types
- Streams can be passed into functions as parameters (and/or returned).
  - Because of how the stream classes were set up, they can only be passed by reference, however

Passing Stream Objects into Functions

- So, for instance, the following can be return types or parameter types in a function:

```
ostream &
istream &
ofstream &
ifstream &
```

- Why? - Functions that produce output are more flexible when they can direct their output to different destinations

Passing Stream Objects into Functions

- Example of a more limited function:

```
void Show(){
   cout << "Hello, World\n";
}
```

- A call to this function always prints to standard output (`cout`).

Passing Stream Objects into Functions

- Same function, more versatile:

```
void Show(ostream& output)
{
   output << "Hello, World\n";
}
```

- We can do the printing to different output destinations now:

```
Show(cout); // prints to standard output stream
Show(cerr); // prints to standard error stream
```

Passing Stream Objects into Functions

- This works with file stream types, too:

```
void PrintRecord(ofstream& fout, int acctID, double
    balance)
{
   fout << acctID << balance << '\n';
}
```

Passing Stream Objects into Functions

- Now, we can call this function to print the same data format to different files:

```
ofstream out1, out2;

out1.open("file1.txt");
out2.open("file2.txt");

PrintRecord(out1, 12, 45.6); //print to file1
PrintRecord(out1, 124, 67.89); // print to file1
PrintRecord(out2, 100, 123.09); // print to file2
PrintRecord(out2, 11, 287.64); // print to file2
```