Ve 280

Programming and Introductory Data Structures

I/O Streams; Testing

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

Input/Output

Streams

- A popular model for how input and output is done in computer systems is centered around the notion of a **stream**.
- A stream is just a sequence of data with functions to put data into one end, and take them out of the other.

```
cin >> a;
```

Input/Output

Streams

- Typical streams:

```
keyboard   →   program
display    ←   program
file       →   program
file       ←   program
string     →   program
string     ←   program
```

- In C++, streams are **unidirectional**.
- Data is always passed through the stream in one direction.
- If you want to read and write data to the same file or device, you need two streams.

Input/Output

Streams

- In general, there are two kinds of stream data: **characters** and **binary data**.
- Characters are usually used for:
  - Communicating between your program and a keyboard or screen.
  - Reading and writing files.
- In addition to text, files can contain arbitrary **binary** data.
  - It is usually much more efficient than character representation.
  - However, it is hard to understand and debug.

- We'll talk about **character streams** here.

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

Output Stream: cout

```
cout << "Hello, world!\n";
```

- Output to screen.
- The << is called the **insertion operator**, and is used to insert things into the output stream.
  - It knows how to **convert** all of the other standard data types to **characters** before inserting them into the stream.

```
int foo = 42;
cout << foo << endl;
```

  - Can be cascaded

```
cout << foo << " " << bar << endl;
```

Alternate Output Streams

- You can also use the Linux I/O **redirection** facility to move the output end of the screen's stream to a file:

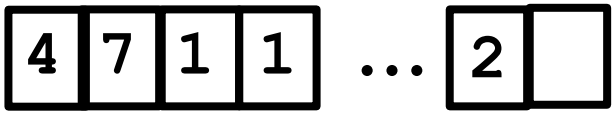
```
$ ./hello > foo
```

- This connects the output end of the `cout` stream to the file "`foo`".

- There is another output stream object defined by the `iostream` library called `cerr`.
- This stream is identical in most respects to the `cout` stream; in particular, its default output is the also screen.
- By convention, programs use the `cerr` stream for **error messages**.

Output: Buffering

- I/O in C++ is **buffered**.
- This means output inserted into an output stream is saved by the underlying operating system (in a region of memory called a **buffer**).



- The content in the buffer is written to the output only when specific actions are taken.

Output: Buffering

- The buffer content is written to the output only when:
  - A newline is inserted into the stream, i.e., **`endl`** or **`'\n'`**
  - The buffer is explicitly flushed. E.g.,

```
cout << "ok" << flush;
```

  - The buffer becomes full
  - The program decides to read from `cin`
  - The program exits
- Once the buffer content is written to the output, the buffer is **cleaned**
- If some content not printed out, may be still in the buffer
- In contrast, output sent to `cerr` is not buffered

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

Input Stream: cin

- `cin >> foo;`
  - Take input from keyboard
  - `>>` is called the **extraction operator**, and is used to extract things from the input stream.
  - Knows how to convert the characters you type into values of simple types and strings.
- Question: what are the variables?

```
int foo;
double bar;
string baz;
cin >> foo >> bar >> baz;
```

Assume inputs is:
`42 3.14 four score\n`

Note: baz is just "four"!

How to get `baz` as "`four score`"?

getline()

- If you need to read strings including whitespace (**blanks**, **tabs**, or **newlines**), use the `getline()` function:

```
cin >> foo >> bar;
getline(cin, baz);
```

Assume inputs is:
`42 3.14 four score\n`

- `getline()` reads all characters **up to but not including** the next newline and puts them into the **string variable**, and then **discards the newline**
- But `baz` is "` four score`"; it keeps the leading space

get()

- The `get()` function reads **a single** character, whitespace or newlines:

```
char ch;
cin.get(ch); // Extracts a character
//from cin stream and stores it in ch
```

- So, we can accomplish what we'd hoped to accomplish by:

```
cin >> foo >> bar;
cin.get(ch);
getline(cin, baz);
```

- This makes `baz` "`four score`".

Assume inputs is:
`42 3.14 four score\n`

The three methods have such different syntax.
However, the three methods can be freely intermixed.

Input: Buffering

- Like `cout`, `cin` is **buffered**.
- Characters typed (which are to be gathered by `cin`) are stored in a buffer **until the enter key** is pressed.
- The characters are then made available to the program as a group.
- This also allows for greater efficiency, and it lets you correct errors before your program sees them (i.e. you can go back and fix something you typed wrong).

Alternate Input Streams

- You can use the Linux I/O **redirection** facility to move the input end of the stream from the keyboard to a file:

```
$ prog < foo
```

- When doing this, remember that the input will not appear on your screen since you did not enter it on the keyboard.
  - This makes funny-looking output, as the input is not echoed.

Failed Input Streams

- The extraction operator will fail if inappropriate data is given to it.
- For example, if:

```
int foo;
cin >> foo;
```

  is presented with:

```
42abc\n
```

  the attempted conversion will succeed, up to the point of the "a", i.e., `foo = 42`

- The stream will be left with "`abc\n`" in it.

Failed Input Streams

```
int foo;
cin >> foo;
```

- However, if you present it with something that **does not** begin with a digit, like:

  `abc`

  then the stream will enter a **failed** state.

- You can test the state of a stream by using it where a bool is expected:
  - For example, `if(cin) {…}` `while(cin) {…}`
  - It returns **true** if it is **good**, false otherwise.

- A failed input stream will resist all attempts to extract more data from it, until you **clear** it via `cin.clear()`.

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

File Streams

- Why use files?
  - Files allow you to store data permanently!
  - Data output to a file lasts after the program ends
  - An input file can be used over and over. No typing of data again and again for testing
- File stream: I/O between file and program

- Linux has I/O redirection facility. Then, why use file streams?
  - E.g., when you need to write to two files

Using File Streams

- `#Include <fstream>`
- Declare an **input** file stream object

```
ifstream iFile;
```

- Declare an **output** file stream object

```
ofstream oFile;
```

- The file stream object must be connected to a file
  - Connecting a stream to a file is opening the file for the stream

```
iFile.open("myText.txt");
```

Must be a C-style string, cannot be C++ string!
use c_str() to convert C++ string into C string

Using File Streams

- Use the **input** file stream: use the extraction operator >> and the `getline()` function

```
int bar;
iFile >> bar;
string baz;
getline(iFile, baz);
```

- Use the **output** file stream: use the insertion operator <<

```
oFile << bar;
```

Closing a File

- After using a file, it should be closed

```
file_stream.close();
```

  - This disconnects the stream from the file.
- Why closing a file?
  - Close files to reduce the chance of a file being **corrupted** if the program terminates abnormally.
  - It is important to close an output file if your program later needs to read input from that output file
- The system will automatically close files if you forget as long as your program ends normally
  - … but **explicitly** closing the file is recommended!

Input File Streams

Example

- Consider the following:

```
#include <iostream>
#include <fstream>
using namespace std;
void main() {
  ifstream iFile;
  int bar;
  iFile.open("foo");
  iFile >> bar;
  cout << "The answer is " << bar << ".\n";
  iFile.close();
}
```

- This opens the file named `foo` for reading, and associates it with the input stream object `iFile`.
- Thereafter you can extract input from the file in the same way we did using `cin`.
- If the file named "`foo`" contains the characters "`42`", this program will output:
  `The answer is 42.`

Failed File Streams

- The file stream enters the failed state if:
  - It cannot be opened.
  - You attempt to read past the end of the file.
- A stream's state may be checked by evaluating the stream object:

```
if(iFile) { ... }
```

  - A stream in the failed state will return false.
- Example

```
iFile.open("a.txt");
if(!iFile) {
    cerr << "Cannot open a.txt\n";
    return -1;
}
```

Example of Reading File

```
while(iFile) {
    getline(iFile, line);
    cout << line << endl;
}
```

Why not good?

How to correct this?

- Normally, after getline reads an entire line, iFile points at the position of the "\n"
- If getline reads the last line, iFile points to the end of file
  - **Note**: iFile is still good! So, the program will issue another getline, which reads nothing
  - So, the program will print an empty line
  - This time, iFile passes the end of the file and loop terminates

Example of Reading File: Correction

```
while(iFile) {
   getline(iFile, line);
   if(iFile) {
      cout << line << endl;
   }
}
```

Example of Reading File

- Another much simpler (and correct) way

```
while(getline(iFile, line)) {
   cout << line << endl;
}
```

- `istream& getline(istream& is,`
  `string& str);`
- Return value: a reference to the its parameter `is` (with the value after issuing the current getline).

- Question: why it works?

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

String Stream

Motivation

- Suppose that you use the `getline()` function to read an entire line from a file and the result is stored in a string.

```
string line;
getline(iFile, line);
```

- Suppose that the line contains an int followed by a double. We want to read these two numbers from the string `line`.

- We can use input string stream!
  - It reads characters in a string and convert them into values of proper types

String Stream

Motivation

- Suppose we have a string of a book name and an int of its published year. We want to create a string whose first part is the book name and the second part is its published year.
  - Notice that we need to convert the int to a string!

- We can use output string stream!
  - It writes to a string
  - It knows how to convert standard data types into characters and insert them into the string

String Stream

- There are two types of string stream: **input** string stream and **output** string stream.

- C++ defines string stream in the sstream library

```
#include <sstream>
```

- Declare an input string stream object

```
istringstream iStream;
```

- Declare an output string stream object

```
ostringstream oStream;
```

Input String Stream

- When we use input string stream, it is usually assigned a string it will read from.

```
iStream.str(a_string);
```

- We can use extraction operator >> on an input string stream to retrieve the data.

```
istringstream iStream;
int foo;
double bar;
iStream.str(line);
iStream >> foo >> bar;
```

If `line` is the string "`42 3.14`", then

```
foo = 42;
bar = 3.14;
```

Output String Stream

- We can use output string stream to format a string.
  - For example, we might have a collection of numeric values but want their string representation.

- We use insertion operator << to insert characters into an output string stream.
- We fetch the string value of the string stream using the member function `str(void)` of a string stream.

Output String Stream

Example

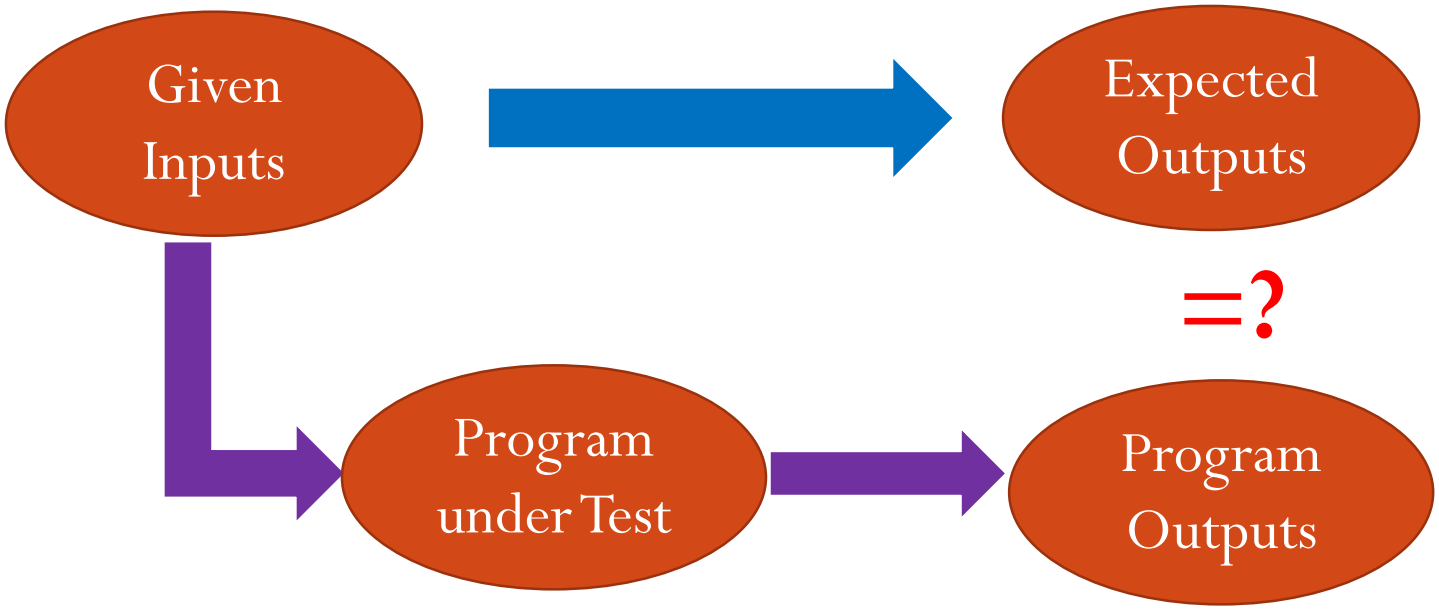
```
int foo = 512;
int bar = 1024;
string result;
ostringstream oStream;
oStream << foo << " " << bar;
result = oStream.str();
```

result is a string "512 1024".

Outline

- I/O Streams
  - Overview
  - Output Stream cout
  - Input Stream cin
  - File Stream
  - String Stream
- Testing

Testing


Given Inputs
Expected Outputs
=?
Program under Test
Program Outputs

Testing

It's important!

- Be skeptical.

- Typically, the difference between a good and bad score on a project doesn't have much to do with your talent as a programmer. **It has much more to do with your talents as a tester!**

- Testing is not the same as debugging
  - Debugging: **Fixing** something once you know it's broken.
  - Testing: **Discovering** that something is broken.

Testing

It's important!

- Some tips and truths about being a good tester:

1. Convince yourself that the code is broken.
2. Be in an adversarial frame of mind.
3. NEVER REST and must ALWAYS BE DILIGENT, because the code is NEVER FINISHED!
4. Everyone makes mistakes, and one essential nature of a mistake is that the person who made it didn't realize it was wrong – you thought it was perfect!

Testing

End-to-end vs. incremental testing

- End-to-end testing is not a good idea
  - Errors made early tend to be pervasive and fixing them requires re-writing a large fraction of the existing program
  - Putting off testing until the program is "finished" increases your workload

- Instead, **test individual pieces of your program (such as functions) as you write them**
  - This is **incremental testing**

Incremental Testing

The better type of testing

- There are two advantages of incremental testing:

1. You are testing smaller, less complex, easier to understand units.
2. You just wrote the code, so you have a firm expectation of what it should do. If it's broken, it is fresh in your mind, so you can more easily fix it.

- This will often require you to write extra code (**the driver program**) to test your program effectively. However, this is usually time well spent.

Five Steps in Testing

- To test some piece of code (either a component or a whole piece):

1. Understand the specification
2. Identify the required behaviors
3. Write specific tests
4. Know the answers in advance
5. Include stress tests

Five Steps in Testing

1. Understand the specification

- For an entire assignment, read through the specification very carefully, and make a note of everything it says you have to do – and stay away from the computer ☺
- Since you have to break down the solution into (smaller) constituent parts, you must write specifications for these parts.

- Sometimes your program as a whole may not work correctly, because you misunderstand the specification.

Five Steps in Testing

2. Identify the required behaviors

- For any specification, boil the specification down to a list of things that must happen.
- These are the "**required behaviors**" and a correct implementation **must exhibit all of them**.

Example: you are asked to write a command-line program called `fact` which takes one argument and calculates the factorial of the argument

Required behaviors

- If there is no argument, output "missing argument"
- If there is more than one argument, just work on the first, ignoring the remaining
- If the argument is not integer, report "non-integral value"
- If it is a negative integer, report "negative integer"
- If it is 0, output 1
- If it is positive integer n, output n!

Five Steps in Testing

3. Write specific tests

- For each of your required behaviors, write one or more test cases that check them.

- To the extent possible, the test case should check **exactly** one behavior – no more!
  - That way, if the case fails, you know where to start looking.

Five Steps in Testing

3. Write specific tests

- There are three classes of test cases that make sense:
  - **Simple inputs**
  - **Boundary conditions**
  - **Nonsense**
- Simple cases are those that are "normal" for the problem at hand.
- "Boundary" cases are at the edges of what is expected, or formed to exploit some detail of implementation.
- "Nonsense" cases are those that are clearly unexpected.

Example: Testing Factorial Function

Assume use `cin` to get the input

- Simple inputs
  - An integer >= 1
- Boundary conditions
  - Value 0
- Nonsense
  - Negative values or non-integer values

Five Steps in Testing

3. Write specific tests: Exercise

- What are examples of these cases for testing the power number in project 2?
  - A positive integer is called a power number if it equals m^n, where m and n are both integers and $n \geq 2$.

- Simple inputs:
- Boundary conditions:
- Nonsense:

Five Steps in Testing

4. Know the answers in advance

- Instead of quickly running test cases and glancing at the output:
  - First write down what you expect to be a correct answer.
- If the result differs in **any** way from what you expected, try to figure out why.
- It's possible that your **expectation** had been wrong…or your **implementation**.
- However, doing this ABSOLUTELY REQUIRES that you understand the specification.
  - If you don't, you will create an incorrect solution that satisfies your incorrect expectation!

Five Steps in Testing

5. Include stress tests

- Once you've tested each individual behavior, it's time to test all of them in concert.
- For this, you want **large** and **long running** test cases.
  - They must be **large**, to exercise resource limits in your program.
    - E.g., some web applications need to be tested under a large amount of simultaneous accesses.
  - They must be **long running**, because some errors are the result of lots of little bugs that individually don't matter much, but as they cascade produces catastrophic results.
    - E.g., the accumulation of the round-off error
    - E.g., the memory leakage

References

- **C++ Primer (4th Edision)**, by *Stanley B. Lippman, Josée Lajoie, Barbara E. Moo*, Addison-Wesley Publishing (2005)
  - Chapter 8.4 File Streams
  - Chapter 8.5 String Streams