# CMPT 115: Principles of Computer Science

## How to do More with Input/Output

Department of Computer Science
University of Saskatchewan

February 8 – 12, 2016

# Laboratory 5 Overview

- Pre-lab Reading
  1. Advanced Console Input/Output (I/O)
  2. File Input/Output
- Exercises (to hand in with Assignment 5)
  - Exercise 4 (page 34)
    File(s): `lab5exercise4.cc`
  - Exercises 5 (page 35)
    File(s): `lab5exercise5.cc`

# Part I

# Pre-Lab 3 Reading

# Console I/O

- Console I/O is input from the keyboard, and output to the computer's display.
- In Eclipse, the console is a special window.
- On the command-line, the console is the terminal application (Terminal on Mac, Konsole on Linux).
- We have used `cin` and `cout` in a very simple manner, but it is possible to do a lot more.
- This lab shows some of these advanced features.

# The concept of buffering

- A *buffer* is memory that is used to store data temporarily.
- Buffers are used in many communications tasks, to help manage differences in the rate at which data can be processed.
- E.g., buffering for streaming video. Video data is accumulated before play starts, so viewing is less affected by minor network delays.
- Our friends `cin` and `cout` use buffering too!

# Understanding `cout` as buffered output

- Output from `cout` is *buffered*.
- The operating system does not actually display data to the screen until `cout` sends `endl`, or if the program terminates.
- This can cause much confusion when debugging.
- Suppose you send data A to `cout`, but do not send `endl`.
- The data A will be waiting in the buffer, but the program will pass to other instructions.
- If your program crashes for any reason, the data A in the buffer does not appear.
- Since data A did not appear, you might think the program crashed *before* data A was sent to `cout`.

# Demonstration

```
// buffering.cpp
// demonstrating buffering: cout

#include <iostream>
using namespace std;

int main() {
  // send data to the console, but no endl
  cout << "cout with endl: buffer displayed" << endl;
  cout << "cout no endl: data stays in the buffer until next endl...

  // now set up some code that will definitely fail eventually
  int data[100];

  // use indices that are way too big for our array!
  for (int i = 0; i < 10000; i++) {
    data[i] = 0;
    if (i > 100) cout << "i = " << i;
  }
  cout << "Just about to exit.";
  return 0;
}
```

# Dealing with `cout` while debugging

- Buffering is very good when you are not debugging.
  - It speeds up communications to the console.
- Buffering can be misleading when debugging.
- When using `cout` for debugging, always end your `cout` calls with a `endl`.
- Alternatively, you may use the error output stream `cerr` for debug messages instead of `cout`.
  - `cerr` is very like `cout`, but it is a separate stream, and usually connected to the console.
  - `cerr` is not buffered. Data goes immediately to the console.

# Understanding `cin` as buffered input

- Input from `cin` is *buffered* too, but differently.
- When called, `cin` first checks if there is any data in its buffer.
- If there is no data in the buffer, `cin` pauses and waits for input from console.
- The user may type a lot of data, or a little.
- The operating system sends *all data up to and including a newline* (end of line, or return key).
- If the user types more data than needed, the extra data is stored in `cin`'s buffer.
- If `cin` is called again, data is taken from the buffer directly, without the user typing anything.
- An activity on page 27 will demonstrate this idea.

# Using `cin` cleverly

- If your program uses `cin` in the standard way, you can send all the data your program needs all in one line!
  - Separate the data with spaces, though.
- Copy/paste your data from a text file
- Or use file redirection.
- Testing your programs would be very tedious if you typed in all the data every time.
- From now on, you should streamline your testing by using file redirection, or by taking advantage of buffered console input!

# Formatting Output

- You can control the way `cout` behaves using *manipulators*.
- Manipulators are commands sent to `cout`.
- To use them, add '`#include <iomanip>`' with the other includes.
- Example: Tell `cout` that you want to display data in a column of width 10:

```
cout « setw(10) « i;
```

- This sets a minimum field width of ten characters.
- If the printed value (the contents of i in this case) has fewer characters/digits than the number specified, it will be padded with blanks so that at least 10 characters are printed. Then `setw` affects the next printed value.
- This is useful to format output in a structured way.

# Setting the precision for cout

- Precision refers to the total number of significant digits displayed for a floating point number.
- To set the precision of a floating point number, we use `setprecision`.
- Example:

```
float z = 2.0030003;
cout « setprecision(4) « z « endl;
```

  Only 4 digits are displayed!
- This setting takes effect until it is changed by another manipulator.

# Setting the decimal places for cout

- Precision refers to the total number of significant digits displayed for a floating point number.
- By default, `setprecision` determines the total number of significant digits (not the number of decimal places)
- To set the number of decimal places, first use `setiosflags(ios::fixed)`, which alters the results of `setprecision` to represent decimal places.

```
cout « setiosflags(ios::fixed) « setprecision(4) « z « endl;
```

- This setting takes effect until it is changed.

# Very manipulators. Much awesome. Wow.[1]

- `setiosflags(ios::fixed)` - sets precision of float to be fixed point
- `setiosflags(ios::scientific)` -write in scientific notation
- `resetiosflags( ios::floatfield )` - resets floating point flags to default values
- `setiosflags(ios::left)` - left justifies text within the specified width
- `setiosflags(ios::right)` - right justifies text within the specified width
- `setiosflags(ios::internal)` - center text within the specified width

[1] Ancient meme is ancient.

# Notes on manipulators

- Manipulators are only for `cout`. Not for `cin`.
- Manipulators can be used on any output stream! Not just the console!
- Don't memorize them. Be aware that they exist, and look them up when you need them.
- You won't be tested on them!

# File Streams

- In this lab, we'll see that we can connect to a file from inside our programs.
- This is called "file I/O" (Input/Output).
- File I/O is abstracted in C++ through the notion of a *file stream*.
- A file stream is an ADT to help manage the task of connecting programs to files.
- Our friends, cin and cout, are actually file streams, but they are connected to the console!
- In C++, reading from and writing to a file will look a lot like using cin and cout.

# More on file streams

- There are two kinds of file streams:
  1. Text streams - stream of characters.
  2. Binary streams - streams of bytes.
- We'll only look at text streams in this course.

# File Streams cont'd

- To connect to a file, programs have to prepare for the connection.
- Files must be *opened* before you can access the file.
- Opening a file creates a connection between your program and the file.
- Files must be *closed* when one is done with them.
- Closing a file disconnects your program from the file.
- Failure to close can cause data to be lost!

# Simple example: Sending data to a file

```
// declare an Output File STREAM variable
ofstream outputFileStream;      // will manage an output stream for us

// open the file named filetowrite.txt so we can send data to it
outputFileStream.open("filetowrite.txt", ios::out);

// check that the file opened correctly
if (outputFileStream.is_open()){
    // sending data to the file is just like sending data to the console
    outputFileStream << "Writing to the file" << endl;

    //close the file
    outputFileStream.close();
}
else {
    cout << "file could not be opened" << endl;
}
```

# File Input in C++

- To read from a file, we send the value `ios::in` to the 'open' function.

```
// declare an input stream with a boring name
ifstream fin1;

// open the file named filetoread.txt so we can get data from it
fin1.open("filetoread.txt", ios::in);

// check that the file opened correctly
if (fin1.is_open())

    // getting data from the file is just like getting data from the console
     int x;
    fin1 >> x;  // assuming that there is an integer at the start of the file

else
     cout << "file is not available for opening" << endl;


//close the stream
fin1.close();
```

# End of File

- Suppose you have an input file stream named `fin1`.
- `fin1` (like `cin`) will read from file until the next whitespace (space or return).
- Reading from the file walks forward through the file to the next piece of data.
- An important concept in reading files is the idea of "end of file."
- This means that there is no more data to read!
- It would be an error to try to read more!

# Detecting End of File: the basics

- Any input stream allows you to check if you've reached the end of file (eof)
- The function `fin1.eof()` will return true if and only if the end of the file has *already* been reached.
  - This function does not behave intuitively.
  - It does not warn you that you are **about** to read past the end of file.
  - It only tells you after you have **already** tried to read past end of file.
  - So you have to be careful how you use it!

# Detecting End of File: C++

The following loop looks fine, but doesn't quite work.

```
while (!fin1.eof()) {
  fin1 >> f;
  sum = sum + f;
}
```

You have to check eof after getting data, to check if the end of file was reached; you can't use eof() just before you hit the end of file. Hint: use cin.eof() immediately after using cin. If true, the value you tried to read was not there! If false, the value you tried to use was successfully read! See Exercise 5!

# Part II

# Lab activities and exercises

# Demonstration

```
// buffering.cpp
// demonstrating buffering: cout

#include <iostream>
using namespace std;

int main() {
  // send data to the console, but no endl
  cout << "cout with endl: buffer displayed" << endl;
  cout << "cout no endl: data stays in the buffer until next endl...

  // now set up some code that will definitely fail eventually
  int data[100];

  // use indices that are way too big for our array!
  for (int i = 0; i < 10000; i++) {
    data[i] = 0;
    if (i > 100) cout << "i = " << i;
  }
  cout << "Just about to exit.";
  return 0;
}
```

# Demonstration of buffered output (`cout`)

ACTIVITY: Run the demonstration on page 25 4 times:

1. Run the example code as given, to see how buffering could be misleading.
2. Change the example by adding `endl` in useful places; don't fix the bug though.
3. Delete the `endl` you added, and then change `cout` to `cerr`; don't fix the bug though.
4. Change the `cerr` back to `cout` but fix the bug!

# Demonstration of buffered input (cin)

```
// inputbuff.cpp
// demonstrating buffering: cin
/* copy paste everything on the line below to the console all at once
     a b c 1 2.7 2 3.1 3 -1.0
*/

#include <iostream>
using namespace std;

int main() {
  int someInts[3];
  char someChars[3];
  float someFloats[3];

  for (int i = 0; i < 3; i++) {
    cin >> someChars[i];  // some input
  }
  for (int i = 0; i < 3; i++) {
    cout << someChars[i] << " ";  // some output
  }
  for (int i = 0; i < 3; i++) {
    cin >> someInts[i] >> someFloats[i];  // more input
  }
  for (int i = 0; i < 3; i++) {
    cout << endl << someInts[i] << " " << someFloats[i] << " ";
  }

  return 0;
}
```

# Exercise 1

ACTIVITY: Copy/paste the code on the next slide into an editor; compile and run. [2]

ACTIVITY: Change the program in various ways to understand what the manipulator does:

- Set the width to 10.
- Set some widths to different values (e.g., some 5, others 10).
- Remove the manipulator on some (or all) lines.

[2] If copying/pasting this code into a text editor, watch out! Sometimes, the single quote symbol ' will cause errors in compilation due to pdf font issues. Just delete those and type ' again into the text editor. If you use TextWrangler, use "Zap Gremlins".

# Code for Exercise 1

```
// Using IO manipulators: setw
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

int main(void) {

        char w = 'd';
        int x = 12;
        int y = 6;
        float z = 2.0;

        cout << setw(5) << w;
        cout << setw(5) << x << endl;
        cout << setw(5) << y;
        cout << setw(5) << z << endl;

        return EXIT_SUCCESS;
}
```

# Exercise 2

ACTIVITY: On the following slide, you'll find a program that reads 4 floats, and prints out a table.

- Compile: `g++ -Wall exercise2.cc`
- Run: `./a.out`
- Change the following, and see what happens:
  1. The column width to 5.
  2. The justification to centered.
  3. The number of significant digits to 3 (not decimal places, significant digits).

# Code for Exercise 2

```
// Using manipulators: more!
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

int main(void) {

        float f1, f2, f3, f4;

        cout << "\nenter a number: ";
        cin >> f1;
        cout << "\nenter a number: ";
        cin >> f2;
        cout << "\nenter a number: ";
        cin >> f3;
        cout << "\nenter a number: ";
        cin >> f4;

        cout << setw(8) << setiosflags(ios::fixed) << setprecision(2)
             << setiosflags(ios::right)<< f1;
        cout << setw(8) << f2 << "\n";
        cout << setw(8) << f3;
        cout << setw(8) << f4 << "\n";

        return EXIT_SUCCESS;
}
```

# File I/O in C++ – Exercise 3

ACTIVITY:

- Redo exercise 1, but send output to a file.
- Imitate some of the concepts in slide 19 !
- Steps:
  1. Add `#include <fstream>`
  2. Create an output stream variable.
  3. Use `open()` to connect the output stream to a file
  4. Use `is_open()` to check the connection worked.
  5. Send data to the file!
  6. Check if the data got to the file correctly!
- Save your new program as `exercise3.cc`.

# Solution to Exercise 3

```
// FIle IO example
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;


int main(void) {

        char w = 'd';
        int x = 12;
        int y = 6;
        float z = 2.0;


        ofstream fout1;
        fout1.open("filetowrite.txt", ios::out );

        if (fout1.is_open()){
                        //opens file "filetowrite.txt"; fout1 is now connected to it
                fout1 << setw(5) << w;
                fout1 << setw(5) << x << "\n";
                fout1 << setw(5) << y;
                fout1 << setw(5) << z << "\n";

                        //write to fout1 just like with cout

                fout1.close();

                }else cout<< "bad file";

        cout << setw(5) << w;
        cout << setw(5) << x << "\n";
        cout << setw(5) << y;
        cout << setw(5) << z << "\n";


        return EXIT_SUCCESS;
}
```

# Exercise 4

ACTIVITY:

- Make a text file called "infile.txt".
- In the file, put an integer on the first line, a float on the second line, a character on the third line, a float on the fourth line.
- Now write a program named `exercise4.cc` that reads all four values from the file, and prints them out to the console.
- Steps:
  1. Include `fstream`
  2. Create an input stream variable
  3. Use `open()` to connect the input stream to your data file input.txt
  4. Use `is_open()` to check the connection worked.
  5. Use the input stream to read the four values.
  6. Send each piece of data to the console, to see if the data was read correctly!

# Exercise 5

ACTIVITY: Create a text file `input.txt` that contains as much text as you like.
ACTIVITY: Write a program named `exercise5.cc` that reads one character at a time from `input.txt` and counts how many times the character 'e' appears in the file. It should work no matter how big the input text file is.
Steps:

1. Start from exercise4
2. Create a loop using `while (!fin1.eof())`
   - The while loop block should read the data one piece at a time.
3. After the end of file is detected, display the number of times 'e' appears in the file.
4. To debug: Send each piece of data to the console, to see if the data was read correctly!

# What to hand in

1. Exercise 4 (page 34)
   File(s): `lab5exercise4.cc`
2. Exercise 5 (page 35)
   File(s): `lab5exercise5.cc`