



Programming Bootcamp 2

COSC2802

DAY 11 Streams (part 1)





Day 11 Workshop Overview

1. Program I/O in C++
2. Overloading Stream Operators



Recap | seen so far

- ▶ We have briefly examined how to go simple I/O to the “terminal”
 - Reading in information that a user types from the “terminal”
 - Writing out information to the user on the “terminal”



Program I/O (general)





Input/Output Sources

- ▶ There are 2 typical sources of input and output for programs
 - **Standard I/O** - aka “the terminal”
 - **File I/O**
- ▶ These are the two we will use
- ▶ Other sources of I/O may include:
 - Network I/O (from TCP/UDP connections)
 - External Computer Devices such as:
 - Camera
 - Microphone
 - Speaker



Standard I/O

- ▶ Standard I/O is the “standard” communication channel between a program and the operating system
- ▶ Generally, the “standard” channel is:
 - Directly connected to the computer’s User I/O
 - That is, the computer keyboard and screen/monitor
 - For programs run through the terminal, the terminal provides the standard keyboard and screen interface
- ▶ Standard I/O is divided into 3 streams:
 - stdin (standard input) - typically the keyboard (via the terminal)
 - stdout (standard output) - typically the screen/terminal
 - stderr (standard error) - typically the screen/terminal, explicitly for error reporting



File I/O

- ▶ File I/O refers to:
 - Anything stored on the physical hard drive (disk) of the computer
 - Physical external storage devices (including network mounted devices)
 - Printers
- ▶ Care must be taken to ensure:
 - The data on the hard drives & external drives is not corrupted
 - Other programs do not modify the files while our program is using them
- ▶ In COSC2802 File I/O will deal with “files” stored on the local computer



Abstracting I/O

Does the exact location of the I/O matter to a program?



Abstracting I/O

Does the exact location of the I/O matter to a program?

▶ No!

- Reading/Writing from files or standard locations is the same concept

▶ I/O is abstracted into *I/O streams*



I/O Streams

► A stream is:

- A method to communicate with any **device**
- A consistent interface for the programmer
- Independent of the actual device being used
- A level of abstraction between the programmer and the device
- Can write to disk file or another type of device (e.g. console)
- Has two types: (1) text streams; (2) binary streams

► A device may be:

- Standard I/O
- Files (local & on external hard drives)
- Network connections
- External devices



Binary Streams

► Binary Streams

- A sequence of bytes (1's and 0's)
- No character translation occurs
- There is a 1-to-1 correspondence between bytes of the stream and the actual device
- May contain a certain number of null bytes at the end
 - For example, for padding so the file fills a sector on a disk

Why is binary useful?

- Compact/efficient representation of complex data objects
- Serialisation of memory e.g. to linear sequence of bytes
 - e.g. saving of game state
- It is how computers actually store and transmit data.



Text Streams

► Text Streams

- A sequence of characters
- Can be organised into lines terminated by a newline character
 - Optional for the last line
- Character translation may occur as required by the host environment
- For example:
 - newline → carriage return / linefeed [when writing]
 - Carriage return / linefeed ← newline [when reading]
- Not necessarily a 1-to-1 relationship between characters of the stream and the actual device



Program I/O in C++





Creating and Writing Output Stream (ostream)

Output stream object (e.g. `std::cout` or `std::ofstream`) can be written to:

1. Writing to Console/Standard output:
 - **`std::cout`** is default `std::ostream` associated with console/standard output
2. Writing to a File:
 - **`std::ofstream`** create this stream to write to file
3. Writing to a String:
 - **`std::ostringstream`** write to in-memory string (concatenate string together)

In all cases output is via the **insertion operator** (`<<`)

- Predefined in C++ for use with output streams

Standard I/O | C++ STL (cout)

- For output, use the `cout` object
 - Contained in the `<iostream>` header
 - Within the `std` namespace
- Uses the output operator (`<<`)

`<output location> << <what to output>`

- Uses default formatting for output
 - Returns a value - the output location
 - Allows operators to be chained
- Example

```
std::cout << 7 << 'a' << 4.567 << std::endl
```



2. Writing to a File

- ▶ To write to a file, a file output stream is required
 - `std::ofstream` class
 - Found in `fstream` header file
- ▶ Similar to using `std::cout`, except:
 - Before writing, must **open** the file (for writing)
 - When opening a file, provide the file name
 - When done, must **close** the file
 - This is so your OS knows you are finished using the file
- ▶ When opening, there are two modes
 - Normal - creates a new file, erasing any existing file with the same name
 - Append - add to the end of an existing file

2. Writing to a File

- Normal: Opening to WRITE to FILE (create or overwrite existing)

```
std::string filename("file.txt"); ← Filename - Relative Path
std::ofstream outFile; ← File output stream
outFile.open(filename); ← Open for normal output
outFile << "Writing to File" << endl; ← Write
outFile.close(); ← Close file
```

- Append: Opening to add to file

```
outFile.open(filename, std::ofstream::app); ← Open for Appending
outFile << "Appending to File" << endl;
outFile.close();
```



3. Writing to String (zybooks)

Writing to in-memory using Output String Stream

Useful when:

- Want to insert characters into string buffer (not to screen/standard out)
- If you want to format strings in a nice way before outputting them.

Create output string stream variable of type ***ostream***

- can insert characters into an ***ostream*** buffer using <<
- ***ostream*** is derived from ***ostream***.
- Can use ***ostream*** same as ***cout*** stream

Example zybooks (Output String stream)

```
std::ostringstream infoOSS;    // Output string stream
std::string infoStr;           // Information string
std::string firstName;         // First name
std::string lastName;          // Last name
int userAge;                   // Age

// Prompt user for input
std::cout << "Enter \"firstname lastname age\": " << std::endl;
std::cin >> firstName;
std::cin >> lastName;
std::cin >> userAge;

// Write user input to string stream
infoOSS << lastName << ", " << firstName;
infoOSS << " " << userAge;

// Appends (minor) to string stream if less than 21
if (userAge < 21) {
    infoOSS << " (minor)";
}

// Extract string stream buffer as a single string
infoStr = infoOSS.str();

std::cout << "Information: " << infoStr << std::endl;
```

Checking status of Output Streams

- ▶ It may be necessary to check the “status” of the output stream
 - The `OutputStream` class has methods to do this
- ▶ It may also be necessary to “flush” the stream to ensure the contents is written
 - Typically a stream automatically flushes after every newline

<code>good()</code>	Check whether state of stream is good
<code>fail()</code>	Check whether either failbit or badbit is set
<code>flush()</code>	Flush output stream buffer

i.e. some error occurred (e.g., corrupted file/memory)

Flushing may be necessary for real time reading/writing between programs. Normally it is done when a program finishes running, but if multiple programs are reading a file while it is being written, a flush may be necessary to ensure the file is ready to be read.



Creating and Reading Input (istream)

Input stream object can read input from:

1. Reading from Console/Standard input:
 - **`std::cin`** is default `std::istream` associated with console/standard input
2. Reading from a File:
 - **`std::ifstream`** create this stream to read from file
3. Reading from a String:
 - **`std::ostringstream`** read input from in-memory string (treat string as input source and extract data as from console and/or file)

In all cases input is via the **extraction operator (>>)**

- Predefined in C++ for use with input streams

Standard I/O | C++ STL (cin)

► For input, use the `cin` object

- Contained in the `<iostream>` header
- Within the `std` namespace

► Uses the input operator (`>>`)

```
<input location> >> <variable>
```

- This is context sensitive!
- Uses the type of the input variable to determine what to read from input
- (Can also be chained)

► Example

```
int x  
std::cin >> x
```



Standard I/O | C++ STL (cin)

► What about:

- End of input?
- Input error or failure?

► `cin` is an object - you should be familiar with these from Java

- Has functions to check for these things
 - `eof()` - check for end of file
 - `fail()` - check for read error

Error checking is large part of Studio 2!

The purpose of error checking is to ensure your program doesn't crash on erroneous input.

E.G.: if your program calculates the addition of 2 integers, but the user inputs a string. What happens?? You need to be able to handle that (later anyway).

Example: Finding the sum of integers in a file

```
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    std::ifstream inFS;
    std::vector<int> numbers;
    std::string file_name = "day14/numFile.txt";
    int num;

    std::cout << "Opening file: " << file_name << std::endl;

    inFS.open(file_name);
    if (!inFS.is_open()) {
        std::cout << "Could not open file!" << std::endl;
        return EXIT_FAILURE;
    }

    std::cout << "Reading numbers from file..." << std::endl;
    while (inFS.good()) {
        inFS >> num;
        numbers.push_back(num);
    }

    inFS.close();

    for (auto n : numbers) {
        std::cout << n << ", ";
    }
    std::cout << std::endl;

    return EXIT_SUCCESS;
}
```




Read from File Variations

There are a lot of different variations on reading from files:

Reading Line-by-Line

```
std::string line;
while (std::getline(inFS, line)) {
    std::cout << line << std::endl;
}
```

Reading Word-by-Word

```
std::string word;
while(inFS >> word) {
    std::cout << word << std::endl;
}
```

Reading Char-by-Char

```
char c;
while (inFS.get(c)){
    std::cout << c << std::endl;
}
```



Reading from a **STRING**

Read from string using Input String Stream

Useful when:

- Want to read input data **from a string** rather than from the keyboard

Create input string stream variable of type ***istringstream***

- reads input from an associated string instead of the keyboard (standard input).
- ***istringstream*** is derived from ***istream***.
- Can use ***istringstream*** same as ***cin*** stream

Example | istream (zybooks)

```
#include <iostream>
#include <sstream>
#include <string>
```

Extracting characters up to next white space from a string using >> (similar to with cin)

```
int main() {
    std::string userInfo = "Amy Smith 19"; // Input string
    std::istringstream inSS(userInfo);     // Input string stream
    std::string firstName;                 // First name
    std::string lastName;                  // Last name
    int userAge;                           // Age

    // // (1) Parse name and age values from input string
    // inSS >> firstName;
    // inSS >> lastName;
    // inSS >> userAge;

    // (2) Can chain them
    inSS >> firstName >> lastName >> userAge;

    // Output parsed values
    std::cout << "First name: " << firstName << std::endl;
    std::cout << "Last name: " << lastName << std::endl;
    std::cout << "Age: " << userAge << std::endl;

    return EXIT_SUCCESS;
}
```



Checking the status of Read Streams

- ▶ The “status” of an input stream is checked similar to output streams
- ▶ A special check is if the end-of-input (^D character) is reached

<code>good()</code>	Check whether state of stream is good
<code>fail()</code>	Check whether either failbit or badbit is set
<code>eof()</code>	Check if EOF is reached

