C++ Programming

Input and Output

Input/Output

- In C++, a standard library (`iostream`) provides streaming output and input objects similar to Java's System.out and System.in
- Two commonly used output stream objects:
  - cout: standard output stream (to the terminal, like C's stdout)
  - cerr: standard error stream (also to the terminal, like C's stderr)
  - Even though they both go to the terminal, that can be treated and redirected separately in different ways; also, cout is typically buffered while cerr is not
- One commonly used input stream object:
  - cin: standard input stream (from the terminal keyboard, like C's stdin)

it is a buffer. so it may not print if something wrong happens.

Input/Output

- As an important note, the standard input/output functions in C are still accessible in C++
- This includes `printf()`, `fprintf()`, and so on
- Generally, programmers are encouraged to use C++'s input/output streams in a C++ program, but it is not uncommon to find C's functions in use regardless

Input/Output

- I/O operations in C++ are performed by operators (not methods or functions, per se)
  - Insertion operator (<<) does output
  - Extraction operator (>>) does input
- The direction of the symbols indicates the data's destination:
  - << inserts data on to an output stream (cout or cerr)
  - >> extracts data from an input stream (cin) into a variable

Insertion

- << is called the output or insertion operator
- General syntax:

```
cout <<  expression;
```

- For example:

```
cout << "You are "  << age  <<  " years old" << endl;
```

Extraction

- >> is called the input or extraction operator
- General syntax:

```
cin >> variable;
```

- The only thing that can go on the right of the extraction operator is a variable, as something is needed to store data extracted from the input stream

Insertion and Extraction

- Consider the following example:

```
int x;  // variable declaration

cout << "Enter a number: "; // print a prompt
cin >> x; // read value from terminal into variable x
cout << "You entered: " << x << endl; // echo
```

Limitations of Extraction

- The extraction operator works fine as long as a program's user provides the right kind of data – a number when asked for a number, for example
- If erroneous data is entered (a letter instead of a number, for example), the extraction operator isn't equipped to handle it; instead of reading the data, it puts a premature end to the extraction

Limitations of Extraction

- When using the extraction operator to read input characters into a string variable:
  - The >> operator skips any leading whitespace characters such as blanks and newlines
  - It then reads successive characters into the string, and stops at the first trailing whitespace character (which is not consumed, but remains waiting in the input stream) ← as buffer, and it may be dangerous.
  - You can use this to type check your input better, but the whitespace separation of input is still an issue …

Limitations of Extraction

- For example, consider this code:

```
string name;
cout << "Enter your name: ";
cin >> name;
cout << "You entered: " << name << endl;
```

- If you were to enter “John Doe” at the prompt, the variable `name` would only receive “John” and not the full “John Doe”

and "_Doe" is in input buffer, waiting for next input

this is a problem!

Limitations of Extraction

- Let's try this again:

```
string name;
cout << "Enter your name: ";
getline(cin, name);
cout << "You entered: " << name << endl;
```

then "John Doe" is entered in, including ending char

- If you were to enter "John Doe" at the prompt, the variable name now receives the full "John Doe"

Limitations of Extraction

- If we wanted to parse a line and extract and type check data from it, there are various methods in the `string` class to assist us
- Another option would be to use something called a `stringstream` that creates a stream-like interface to a `string`, allowing us to use the extraction operator on it in a smarter way
- Once we see discuss classes more in a bit, we can see how to do this sort of thing a bit better …

File Input/Output

- When reading data from a file, the programmer doesn't need to be concerned with interacting with a user

- This means prompts are unnecessary, and more than one piece of data can be extracted from the input stream and moved around using a single line of code

File Input/Output

- Earlier, we used the `iostream` standard library, which provides cin and cout as mechanisms for reading from/writing to standard input and standard output respectively

- We can also read from/write to files; this requires another standard C++ library called `fstream`

File Input/Output

- The `fstream` library provides the following classes to perform output and input of characters to/from files:

- `ofstream`: Stream class to write on files
- `ifstream`: Stream class to read from files
- `fstream`: Stream class to both read and write from/to files

File Input/Output

- As in C, in C++ you generally do the following to work with files:
  - Declare your variable for handling the file
  - Open the file using this variable, naming the file and (as necessary) the type of operation you will be performing on the file
  - Carry out operations (including >> and <<) and call functions to move data to/from the file
  - Close the file when done

File Input/Output

- Consider this for example. What will this do?

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   string line;
   ifstream file;
   file.open("text.txt");
   while (getline(file, line)) {
      cout <<  line << endl;
   }
}
```

read text.txt
print all lines.

for input, you must explicit close the input stream before