WEEK 10

I/O refinements

OVERVIEW

- Week 10-1
  - Stream Objects
    - Input / Output Objects
  - Manipulations
  - States and Robust Validation
  - File Stream Classes
  - Odds and Ends

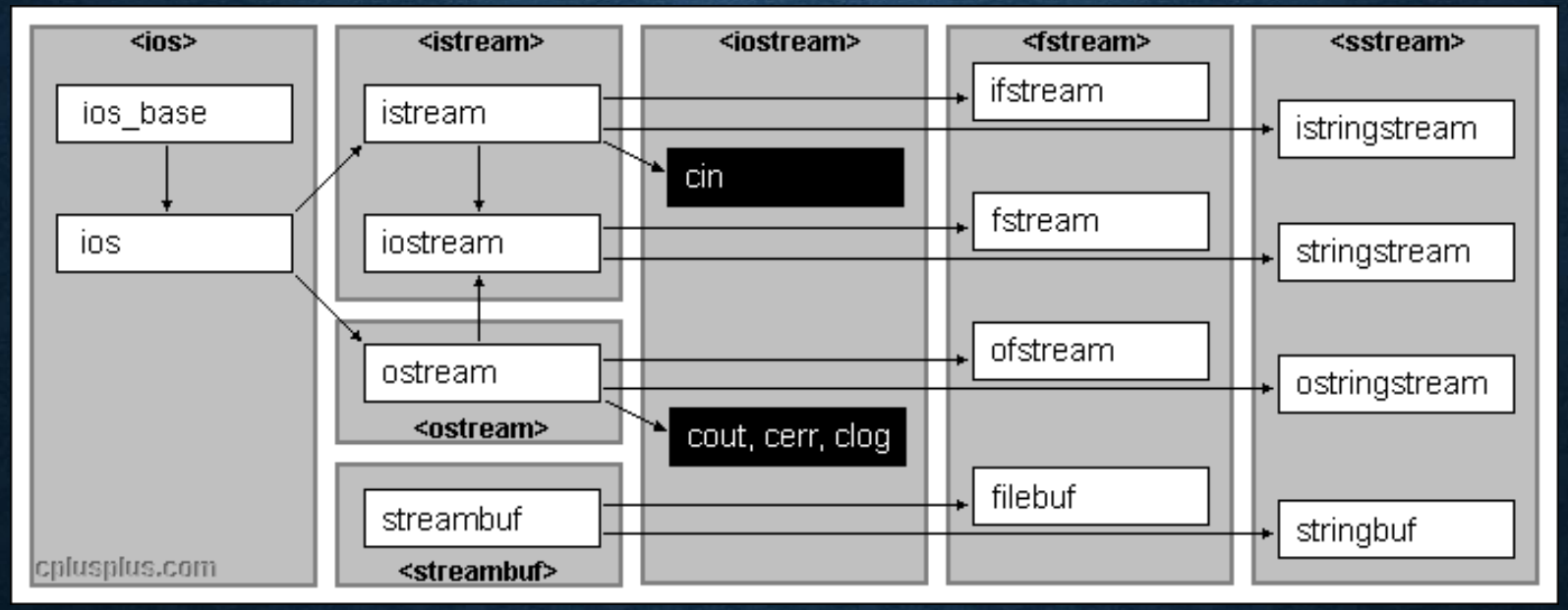
STREAM CLASSES

STREAM CLASSES

- Working with input and output in C++, it works with a notion of utilizing well encapsulated objects (cin, cout) and extraction/insertion operators (<<, >>)
- Whether working with the standard input/output or with files, the approach to working with either is largely similar (we take to again well encapsulated objects and the respective operators)

STREAM CLASSES

The I/O related pieces in C++ are built in the form of a hierarchy of classes.



As can be seen, the stream classes are linked via this hierarchy. The familiarity of working with istream/ostream and filestreams is due to this relation.

INPUT / OUTPUT OBJECTS

INPUT OBJECTS

- Instance of an istream class
- Represents an input device (e.g. keyboard or other peripheral)
- Converts the sequence of characters in the stream to a value stored based on the type of the right-hand side's operand
- Appends nullbyte
- Skips leading whitespace from inputs
- Uses whitespace as a delimiter between inputs

INPUT OBJECTS

Example Usage:

```
char c;
cin >> c;
```

Simple use

```
cin >> i >> x >> c >> s >> z;
```

Cascading use


```
cin >> i;
cin >> x;
cin >> c;
...
```

Cascading
broken down

INPUT OBJECTS

Dealing with whitespaces

```
char str[11];
cout << "Enter a string with leading
whitespace: " << endl;
cin >> str; // Enter something with leading
whitespace
cout << "|" << str << "|" << endl
```

Given an input of “ aaa” for the str variable, what’s the output?

```
// White space as delimiter
cout << "Enter a string with whitespace all
around: " << endl;
cin >> str;
cout << "|" << str << "|" << endl;
```

Given an input of “ aaa ” for the str variable, what’s the output?

INPUT OBJECTS

Cascading inputs and overflow

```
// Cascading inputs and overflow
int i;
char c;
double x;
char s[8];
cout << "Enter an integer,\n"
"a character,\n"
"a floating-point number and\n"
"a string : " << flush;
cin >> i >> c >> x >> s;
cout << "Entered " << i << ' '
<< c << ' ' << x << ' ' << s << endl;
```

If the user inputted a sequence: 1a2.2bluefantasy

What would occur in the following cout line?

Is there a problem?

INPUT OBJECT MEMBER FUNCTIONS – GET()

- get() – extracts a single character or a string from the input buffer
- 3 overloads:
  - get() – extracts a single character
  - get(destination, size) – extracts up to size - 1 characters and adds a nullbyte
  - get(destination, size, delimiter) – same as above but incorporates a delimiter
- Does not skip leading whitespace
- The delimiter is left in the buffer

INPUT OBJECT MEMBER FUNCTIONS – GET()

Example Usage:

```
// Get example
cout << "Enter csv input: ";
char str[25];
cin.get(str, 25, ',');
cout << "Data start " << "|" << str << "| data end" << endl;
char x = cin.get();
cout << "Anything left in the buffer: " << x << " end" << endl;
```

What is the result?

cin.get() obtains input from the standard input stream without the use of the >> operator, it can also specify the number of characters 'gotten' and a delimiter

INPUT OBJECT MEMBER FUNCTIONS – GETLINE()

- getline() – similar to get() but extracts the delimiting character from the buffer
- 2 overloads:
  - getline(destination, size) – extracts up to size - 1 characters and adds a nullbyte
  - getline(destination, size, delimiter) – same as above but incorporates a delimiter

INPUT OBJECT MEMBER FUNCTIONS – GETLINE()

Example Usage:

```
// Getline example
cout << "Enter csv input: ";
char str2[25];
cin.getline(str2, 25, ',');
cout << "Data start " << "|" << str2 << "| data end" << endl;
char z = cin.get();
cout << "Anything left in the buffer: " << z << " end" << endl;
```

What is the result?

cin.getline() operates very similarly to the get() function with one main difference

INPUT OBJECT MEMBER FUNCTIONS – IGNORE()

- ignore() – Ignores / discards characters from the input buffer
- Two overloads:
  - ignore() – Discard a single character.
  - ignore(size, delimiter) – Discards size number of characters or up to the delimiter
- Default delimiter is the end of file character (EOF)

INPUT OBJECT MEMBER FUNCTIONS – IGNORE()

Example usage:

```
char first, last;

// Ignore example
cout << "Please, enter your first name followed by your surname: ";
first = std::cin.get(); // get one character
std::cin.ignore(256,' '); // ignore until space
last = std::cin.get(); // get one character
cout << "Your initials are " << first << last << '\n';
std::cin.ignore(256,'\n'); // ignore characters till new line
```

OUTPUT OBJECTS

- Instance of the ostream class
- Represents an output device (e.g. a terminal window, a file…)
- Converts the data in its right operand into a sequence of characters based on the type of the operand

OUTPUT OBJECTS

- There are three distinct standard output objects provided by the ostream class:
  - cout – transfers a buffer sequence of characters to the standard output device
  - cerr - … standard error output device
  - clog - … standard log output device

OUTPUT OBJECTS

Example usage:

```
char x = 'a';
cout << x;
```

Standard usage

```
cout << x << y << z << "hello" << "world" << endl;
```

Cascaded usage

```
cout << x;
cout << y;
cout << z;
cout << "hello"
```

Cascading
broken down

OUTPUT MEMBER FUNCTIONS

Functions

- width(int) – sets the field width to the integer received
- fill(char) – sets the padding character to the character received
- setf(…) – sets a formatting flag to the flag received
- unset(…) – unsets the flag received
- precision(int) – sets the precision to the integer received

Examples

- cout.width(10);
- cout.fill('*');
- cout.setf(ios::fixed);
- cout.unset(ios::fixed);
- cout.precision(2);

MANIPULATORS

MANIPULATORS <IOMANIP>

- While the input and output objects have member functions that allow for more precise capturing of data or a specifically formatted output of text, they fall into being used outside of the insertion and extraction operators:

```
double pi = 3.141592653;
cout << "1234567890" << endl;
cout.setf(ios::fixed);
cout.width(10);
cout.precision(2);
cout << pi << endl;
```

Falls outside of the ‘stream’

MANIPULATORS <IOMANIP>

- Through using the <iomanip> library we can have access to inlining these stream modifying member functions into the sequence of extraction/insertion operators as though they were arguments to be fed into a stream:

```
double pi = 3.141592653;
cout << "1234567890" << endl;
cout.setf(ios::fixed);
cout.width(10);
cout.precision(2);
cout << pi << endl;
```



```
#include <iomanip>
...
double pi = 3.141592653;
cout << "1234567890" << endl;
cout << fixed << setw(10) <<
setprecision(2) << pi << endl;
```

INPUT MANIPULATOR EXAMPLE

```
int main( ) {
char a[5], b[2], c, d[7];
cout << "Enter : ";
cin >> setw(5) >> a >> // setw sets the field width for the next string input
setw(2) >> b >> noskipws >> // noskipws turns off skipping leading whitespace
c >> skipws >> d; // skipws turns on skipping whitespace
cout << "Stored '" << a <<
"' & '" << b <<
"' & '" << c <<
"' & '" << d << "'" << endl;
}
```

STATES AND ROBUST VALIDATION

STATES

- The ios base class has functions that can report or change the state of istream and ostream objects and this include:
  - good() – the next operation might succeed
  - fail() – the next operation will fail
  - eof() – the end of file or data has been encountered
  - bad() – the data may be corrupted or the stream's integrity has been lost
  - clear() – resets the state to good
- The use of these states can allow for the more robust input processing

STATES

Example usage:

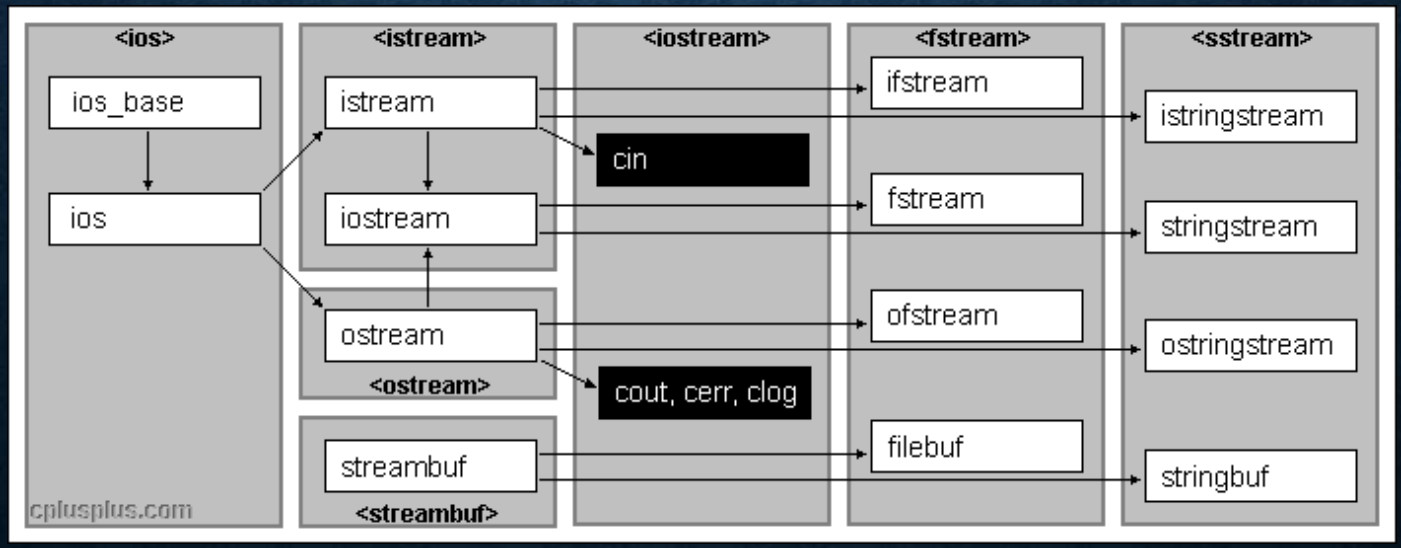
```
int value;
cin >> value;

if(cin.fail()) { // checks if cin is in a failed state
  cin.clear(); // clears state to allow further extraction
  cin.ignore(2000, '\n'); // clears the input buffer
}
```

FILE STREAM CLASSES

FILE STREAM CLASSES

- File streams are managed by mainly the fstream classes which derives from iostream.
- The fstream classes include:
  - ifstream – processes input from a file stream
  - ofstream – process output to a file stream
  - fstream – processes both input and output



FILE OPEN-MODE FLAGS

- When opening a file you can specify a flag that determines the connection to the file
  - std::ios::in – open for reading
  - std::ios::out – open for writing
  - std::ios::app – open for appending
  - std::ios:trunc – open for writing but truncate if file exists
  - std::ios::ate move to the end of the file once the file is open
- These flags can be used in combinations
- Example:

```
fs.open("test.txt", std::ios::in) // opened test.txt for reading but not writing

fs.open("test.txt", std::ios::in|std::ios::out) // opened test.txt for reading and writing (default)
```

LOGICAL NEGATION OPERATOR

These two forms are equivalent

```
if (fin.fail()) {
std::cerr << "Read error";
fin.clear();
}
```



```
if (!fin) {
std::cerr << "Read error";
fin.clear();
}
```

The negation operator ! is overloaded to allow for a more streamlined way to check if a file stream is in an error state via a Boolean value.

REWINDING A FILE CONNECTION

- As we traverse through the file either by reading from it or writing data to it, there is a notion of positioning (row, column)
- Much like the cursor you see in a text editor as you type away and erase text, there is a cursor that moves along when you interact with a file stream

REWINDING A FILE CONNECTION

- This cursor can be moved and repositioned via some file stream member functions:
  - istream& seekg(streampos pos) - sets the current position in the input stream to pos
  - ostream& seekp(streampos pos) - sets the current position in the output stream to pos

REWINDING A FILE CONNECTION

Example usage:

```
// position in output stream
#include <fstream> // std::ofstream
int main () {
std::ofstream outfile;
outfile.open("test.txt");
outfile.write("This is an apple",16);

long pos = outfile.tellp();
outfile.seekp(pos-7);
outfile.write(" sam",4);
outfile.close();

return 0;
}
```

seekg(0) - sets the current position in the input stream to 0

seekp(0) - sets the current position in the output stream to 0

tellp() gets the current position of the cursor in the output stream, tellg() is the equivalent for input