CS 213 -- Lecture #12

"Late Night Guide to C++"
Chapter 8 pg 207 - 216
STREAMS
(and other miscellaneous topics)

Administrative...

- Assignment #6 due on Friday, 10/8
  - Anyone who wants theirs returned before the prelim on 10/14, please contact me.

*void and void **

- Although we've been using them, we've never formally talked about them.
- `void` is used to neatly specify that no return value is required
- can also be used to specify that a function takes no parameters

```
// doNothing() is a function which takes no parameters and
// returns no value
void doNothing(void)
{
  int x = 1;  // well, something, but really nothing :-)
}
```

- You cannot create a variable of type `void`.
- That's because it really isn't a type--the compiler would have no idea how big a "void" is.
- You can, however, create a variable of type `void *`...

*void and void * (cont)*

- A `(void *)` type is a pointer to *anything.*
- That is, you can assign any pointer to a variable of type `(void *)`.
- The reverse is not true however.
- You cannot assign a `(void *)` variable to a pointer variable (except another `(void *)`) without explicit type casting.

```
int main()
{
  char *foo = "This is a test";  // Did you know this is legal?
  void *somePtr;
  somePtr = foo;
  foo = (char *) somePtr;
}
```

- So why would you use a `(void *)` anyway?

*void and void * (cont)*

- In short, use a (void *) anytime you need to deal with a pointer of any type.
- Usually, this is as a parameter to a function.
- Consider the hex dump example from the book.
- A low level function to do a hex dump should be able to take any pointer:

```
void hexDump(void *ptr, long size)
{
  char *p = (char *)ptr;
  for (int j=0; j<size; j+=16)
  {
    cout << hex << (unsigned long)p+j << ": ";
    for (int k=j; k<j+16 && k<size; k++)
      cout << hex << (unsigned char) p[k] << " ";
    cout << endl;
  }
}
```

Streams

- Well, here we are, half way through the class (well, almost) and we haven't even spoken about file I/O.
- Until now, that is…
- In the past we've mentioned that when using `cin` and `cout`, we're actually dealing with a *stream* of characters.
- We use the same type of streams to do file I/O
- Let's start with the stream used to write to a file.
- It is called ofstream (for output file stream)
- It is used like this:

```
int main()
{
  ofstream outStream("output.dat");  // name of file to open
  outStream << "This is a test" << endl;
}
```

ofstream

- If we are calling a constructor to open the file, how do we make sure the file was opened?
- Use the `ofstream::is_open()` member function.

```
int main()
{
  ofstream outStream(“output.dat”);  // name of file to open
  // Make sure we actually opened the file!
  if (!outStream.is_open())
  {
    cerr << “Error opening file output.dat” << endl;
    return -1;
  }
  outStream << “This is a test” << endl;
}
```

ifstream

- For reading files in, we use a similar class and syntax.
- The class is called `ifstream`, and is used like this:

```
int main()
{
  ifstream inStream(“input.dat”);  // name of file to open
  // Make sure we actually opened the file!
  if (!inStream.is_open())
  {
    cerr << “Error opening file input.dat” << endl;
    return -1;
  }
  string s;
  inStream >> s;
  cout << “First string in input.dat is: ” << s << endl;
}
```

ifstream & ofstream

- Having defined input and output streams, you might think we have everything we need to copy one file to another.
- Consider the following code:

```
int main()
{
  ifstream in("input.dat");
  if (!in.is_open())  return -1;  // Shortened for space
  ofstream out("output.dat");
  if (!out.is_open()) return -1;  // Shortened for space

  // Copy the file, one string at a time
  while (in && out)  // an “idiom”, means loop until EOS or error
  {
    string s;    in >> s;    out << s;
  }
  cout << "All done!" << endl;
  return 0;
}
```

Demonstration #1

Simple File Copy

*streambuf **

- Oh yeah, now I remember, we have that whole whitespace issue to contend with as well.
- There’s a different way to copy one file to another.
- All streams are derived from the `ios` class, which manages the actual buffer used to store data relative to the stream.
- You can get at a pointer to this buffer.
- You can take advantage of the fact that, for streams, the << operator is overloaded to take a `streambuf *` type…
  - all data in the buffer pointed at by `streambuf *` is sent at the current stream’s buffer.

```
int main()
{
  ifstream in(“input.dat”);
  ofstream out(“output.dat”);
  out << in.rdbuf();  // returns a ptr to in’s stream buffer
  return 0;
}
```

Demonstration #2

Copy File (using streambuf)

ostringstream

- A stream of type ostringstream is used to treat a string like an output stream.
- Any stream operations will modify the string instead of being printed to the console or a file.
- For those who know C, this is similar to `sprintf`().

```
void main()
{
  ostringstream oss;
  int k = 1;
  oss << “This is a test of “ << k << “use of ostringstream” <<
         endl;

  // Note we need to use ostringstream::str()
  cout << “Our string is: “ << endl << oss.str() << endl;
  return 0;
}
```

Demonstration #3

`ostringstream`

istringstream

- Similar to `ostringstream`, we can use a stream of type `istringstream` to apply stream operations to a string buffer.
- Consider the following code:

```
int main()
{
  istringstream iss("5.54 is your change!");
  float f;
  char c1,c2,c3,c4;
  iss >> f >> c1 >> c2 >> c3 >> c4;
  cout << "f is " << f << endl;
  cout << "c1 is " << c1 << endl;
  cout << "c2 is " << c2 << endl;
  cout << "c3 is " << c3 << endl;
  cout << "c4 is " << c4 << endl;

  return 0;
}
```

Demonstration #4

`istringstream`

Overloading <<

- At the end of last lecture we learned that attempting to overload the (`int`) cast of `MyString` had an unfortunate side effect.
- If we overloaded the (`int`) cast and subsequently tried to print out an instance of the `MyString` class like this:
  - `cout << aStr`
- We’d get a zero printed out instead of the string!
- There a way to still allow the (`int`) typecast and not get tripped up by this!
- We can overload the `<<` operator.
- It looks something like this:

```
inline ostream& operator<<(ostream &os,MyString &myStr)
{
  os << (string) myStr;
  return os;
}
```

Overloading << (cont)

```
inline ostream& operator<<(ostream &os,MyString &myStr)
{
  os << (string) myStr;
  return os;
}
```

- As with most binary operators, `<<` must be overloaded globally.
- It takes an output stream reference (`ostream &`) as first argument.
- It takes a reference to whatever type you wish to overload the operator for as the second argument
  - in this case, a MyString reference (`MyString &`)
- You need to return an ostream reference (`ostream &`) which is usually going to be the first parameter.
  - Allows chaining, such as `cout << “aStr is “ << aStr << endl;`
- Now, we can overload the (`int`) cast again and not mess up `cout`!

Demonstration #5

Overloading `<<`

Overloading >>

- Overloading the >> operator is a little trickier because you need to deal with an input buffer as an “array of characters”.
- Fortunately, the `istream` class has some member functions to help.
- The problem is that these usually deal in C-style strings.
- To tackle the problem of overriding >> for `MyString`, think about how we implemented `MyString::readString()`.

```
int MyString::readString(int maxSize)
{
  // Make sure we have enough storage
  if (!growStorage(maxSize))
    return 0;

  cin.getline(storagePtr,maxSize);
  stringLength = strlen(storagePtr);
  return stringLength;
}
```

Overloading >> (cont)

- In MyString::readString() we have the luxury of allowing the user to specify the largest string to be allowed as input.
- Since the >> operator doesn’t allow for an extra argument such as this, we must restrict ourselves to only allowing a string that will fit into the currently allocated memory space for this `MyString` instance.
- You would think that the `allocatedSpace` member variable would be perfect for telling us this information.
- The problem is that >> must be overridden globally, and `allocatedSpace` is a private member variable.
- How can we access this information from over overridden operator?
- Defining a simple getter for `allocatedSpace` is the answer!
- Once we have that in place, we can define our overloaded >> as follows:

Overloading >> (cont)

```
inline istream& operator>>(istream &is,MyString &myStr)
{
  int allocatedSpace = myStr.getAllocatedSpace();
  char *tempBuf = new char[allocatedSpace]; // allocate temp
  is.get(tempBuf,allocatedSpace-1);  // get() retrieves chars
  string tempStr = tempBuf;          // convert to C++ string
  myStr.setValue(tempStr);           // Utilize setter
  delete [] tempBuf;                 // delete temp mem
  return is;                         // return stream
}
```

- Let’s make sure this works...

Demonstration #6

Overloading `>>`

Final Thoughts

- Prelim #1 10/14 in class