Late Developer
Random thoughts of an old C++ guy

C++ Stringstreams

November 16, 2011
Introduction

Anyone who has ever written a C++ program is probably somewhat familiar with the C++ stream input and output facilities. Most programmers start by writing programs like this:

```
int main() {
  int x;
  cin >> x;
  cout << "You entered " << x << endl;
}
```

They then graduate to using the ofstream and ifstream classes to do similar things with file streams. However, many programmers never encounter the **stringstream** classes which parallel the file streams. This article provides an introduction to stringstreams, and describes some of their common uses.

Stringstreams 101

The **stringstream** classes can be thought of as being very similar to to the **istream** and **ostream** classes that you are used to using, such as **cin** and **cout**, except that you have to create them yourself, and they read and write to and from strings held in memory rather than to or from the console, or to or from disk files. Once your program ends, these strings of course go with it, so stringstreams are only useful for creating temporary objects.

The two stringstream classes used in this tutorial are **istringstream**, which you can read from, and **ostringstream** which you can write to. Both classes are declared in the Standard Library header file **<sstream>**. Note that there is a similarly named header **<strstream>**, which implements streams based on C-style arrays. This header is obsolete, has been deprecated by the C++ Standard, and should never be used.

To use an **istringstream**, you populate it in its constructor, and then read from it with the usual stream reading methods:

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  istringstream is( "the quick brown fox" );
  string s;
  while( is >> s ) {
    cout << s << endl;
  }
}
```

This prints out:

```
the
quick
brown
fox
```

For the **ostringstream**, you write to it using the standard stream methods, and then use the special **str()** member function to retrieve everything that was written to the stream as a single string:

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  ostringstream os;
  os << "the ";
  os << "quick ";
  os << "brown ";
  os << "fox";
  string s = os.str();
  cout << s << endl;
}
```

which outputs:

```
the quick brown fox
```

All of the stream features you may be used to using, such as manipulators, may be used on ths stringstream classes, in exactly the way you would use them on the more commonly used stream types.

So that in a nutshell is the stringstream classes – the rest of this article looks at (hopefully) interesting things you can do with them.

Things To Strings

Ever wanted to convert the value in a double variable to a string? Or to be able to write code like this:

```
double d = 123.45;
string s = "The value of d is " + d;
```

As things stand, you can't do it because **operator+()** does not (and cannot) supply a suitable overload. But if you had some way of converting the variable **d** in the above code into a string, then it would work.

The **osstringstream** class allows you to perform conversions from any type that supports the output **operator<<()** for streams to strings. The code is quite straightforward:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  double d = 123.45;
  ostringstream os;
  os << d;
  string s = "The value of d is " + os.str();
  cout << s << endl;
}
```

Here you write a double out to the **ostringstream** using the same **operator<<()** that you would use if you were writing it to **cout**. You then grab all the contents of the stream using the stream's **str()** member function. This function returns as a string everything that has been written to the stream, which in this case is the double "123.45". As **str()** returns a string, you can use **operator+()** to concatenate it with the string literal.

Simple, yes. But a bit unwieldy, particularly if you want to do a lot of conversions. The obvious solution is to write a function, and you might well come up with something like this:

```
string DoubleToStr( double d ) {
  ostringstream os;
  os << d;
  return os.str();
}
```

so that you can then say things like this:

```
int main() {
  double d = 123.45;
  string s = "The value of d is " + DoubleToStr( d );
  cout << s << endl;
}
```

This works, but what if you want to convert ints? Or longs? Or some class of your own devising that supports **operator<<()**? You do not want to have to write a new function for each type, particularly when each function will look exactly the same, except of the type of thing being converted.

When you find yourself in this situation, you should be thinking "templates", and it is very easy to convert **DoubleToStr** to a function that will convert any type that supports **operator<<()** to a string:

```
template <typename T>
string ToStr( const T & t ) {
  ostringstream os;
  os << t;
  return os.str();
}
```

which allows you to write code like this:

```
int main() {
  double d = 123.45;
  int i = 42;
  string s1 = "The value of d is " + ToStr( d );
  string s2 = "The value of i is " + ToStr( i );
  cout << s1 << "\n" << s2 << endl;
}
```

A couple of points about the template function. First, it's a very useful function and one you may be tempted to put in a library header file so you can use it in all your code. This is a good idea, but if you do you must fully qualify the names of all the standard library types with **std::** and not use the **using** directive as in the code above. Secondly, the use of a reference as a parameter is a little inefficient for the built-in types like **int** and **double**, but is necessary if you want it to work efficiently and seamlessly with user-defined types. And thirdly, if you are using the Boost libraries, you already have access to something very similar in the form of the **boost::lexical_cast()** function.

Strings To Things

If you can convert objects to strings using an **ostringstream** and **operator<<()**, it should come as no surprise that you can convert strings to objects using the **istringstream** class and **operator>>()**. Here's a template function that converts a string to any type that supports **operator>>()** and is default-constructible and copyable**.**

```
template <typename T>
T ToType( const string & s ) {
  istringstream is( s );
  T temp;
  is >> temp;
  return ttemp;
}
```

Here, the stream is loaded with the contents of the string **s**. Objects (in this case only a single object) can then be extracted from the stream using **operator>>()** exactly as if they were being read from **cin**. You can then write code like this:

```
int main() {
  string n1 = "123.45", n2 = "42";
  cout << ToType<double>( n1 ) + ToType<int>( n2 ) << endl;
}
```

which prints out **165.45**.

The function described above requires the user to specify the type being converted using the **funcname<type>** syntax, which many consider ugly. It also requires that the type being converted have a default constructor,m so that the temporary object **temp** can be created, and a copy constructor so that its value can be returned. You can get around these problems by re-writing the function:

```
template <typename T>
void ToType( const string & s, T & t ) {
  istringstream is( s );
  is >> t;
}
```

but this then requires that you re-write the calling code to this form, which to my mind is too verbose for real-world use:

```
int main() {
  string n1 = "123.45", n2 = "42";
  double d;
  int i;
  ToType( n1, d );
  ToType( n2, i );
  cout << d + i << endl;
}
```

There is another problem with the **ToType** function – what happens if the conversion cannot be performed? For example, what happens if you say:

```
double d = ToType<double>( "foobar" );
```

You need to detect the conversion failure and indicate an error. In this case it is not possible to indicate an error via the return value, as you don't know what the actual type of that value is going to be, so you must throw an exception. The obvious (to me anyway) exception to throw is **std::bad_cast,** which is declared in **<exception>**:

```
template <typename T>
T ToType( const string & s ) {
  istringstream is( s );
  T t;
  if ( ! (is >> t) ) {
    throw bad_cast();
  }
  return t;
}
```

A good rule to follow when programming is to always try to provide a means of checking if an error would occur, rather than waiting for one to happen and fielding it after the event. So it's a good idea to provide a companion function to **ToType** which checks if the conversion would succeed if it were applied:

```
template <typename T>
bool IsType( const string & s ) {
  istringstream is( s );
  T t;
  return ( is >> t );
}
```

Unit Tests

Another handy use for stringstreams is in unit testing, where their use can simplify the testing of functions that would normally read and/or write to disk files or other streams. For example, assume you have a function like this, which tokenises a text file into a vector of strings:

```
int Tokenise( ifstream & ifs, vector <string> & v ) {
  string token;
  while( ifs >> token ) {
    v.push_back( token );
  }
  return v.size();
}
```

In order to properly test this function you are going to need a number of text files on disk; an empty one, one containing a single token, one with multiple lines, etc. These files have to exists somewhere and have to be version controlled separately from the code they are testing. It would be much nicer if you could write your tests for the function in one self-contained unit. Well, you can do that using stringstreams. You have to change the function so that it takes an **istream** as a parameter, rather than an **ifstream**, but this is good practice anyway:

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

#define TEST( e )      \
  cout << ((e) ? "Passed: " : "Failed: ") << #e << endl

int Tokenise( istream & ifs, vector <string> & v ) {
  string token;
  while( ifs >> token ) {
    v.push_back( token );
  }
  return v.size();
}

int main() {

  vector <string> v;
  istringstream empty( "" );
  TEST( Tokenise( empty, v ) == 0 );
  TEST( v.size() == 0 );
  v.clear();

  istringstream one( "the" );
  TEST( Tokenise( one, v ) == 1 );
  TEST( v.size() == 1 );
  v.clear();

  istringstream lines( "the quick brown fox\njumped over the lazy dog" );
  TEST( Tokenise( lines, v ) == 9 );
  TEST( v[5] == "over" );
  v.clear();
}
```

which produces this reassuring output:

```
Passed: Tokenise( empty, v ) == 0
Passed: v.size() == 0
Passed: Tokenise( one, v ) == 1
Passed: v.size() == 1
Passed: Tokenise( lines, v ) == 9
Passed: v[5] == "over"
```

You can perform all operations on a stringstream that you can on a file stream – for example, you can seek within a stream:

```
#include <sstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  istringstream is( "foo bar" );
  string s;
  is >> s;
  cout << s << endl;
  is.seekg( 0 );
  is >> s;
  cout << s << endl;
}
```

This code prints "**foo**" twice, because after reading it the first time it seeks back to the beginning of the stream using **seekg()** and then reads it again. About the only limitation of stringstreams is that they they are always opened in binary mode and hence will not do newline expansion where this would be performed by file streams in text mode.

Adapting Strings To Streams

Given the implementation of the Tokenise() function described above, it's very simple to write an overloaded function which instead of reading tokens from a stream, reads them from a string:

```
int Tokenise( const string & s, vector <string> & v ) {
  istringstream is( s );
  return Tokenise( is, v );
}
```

This function simply creates a stream from a string and then uses the stream version to perform the tokenisation. This kind of function is called an adaptor, and the stringstream classes are very useful for writing such things. You can of course use ostringstreams when your function needs to write, rather than read.

Conclusion

This article has looked at the C++ stringstream classes, which are in-memory streams that work almost identically to the more familiar file stream classes. These classes are handy for conversion to and from strings, for replacing file streams in tests, and for a number of other purpose, by no means all of which have been covered here!

From → c++, tutorial

Leave a Comment