Linuxtopia

home | library | techotopia | virtuatopia | ebook store

Search

FOLLOW US ON twitter

On-line Guides
- All Guides
- eBook Store
- iOS / Android
- Linux for Beginners
- Office Productivity
- Linux Installation
- Linux Security
- Linux Utilities
- Linux Virtualization
- Linux Kernel
- System/Network Admin
- Programming
- Scripting Languages
- Development Tools
- Web Development
- GUI Toolkits/Desktop
- Databases
- Mail Systems
- openSolaris
- Eclipse Documentation
- Techotopia.com
- Virtuatopia.com

How To Guides
- Virtualization
- General System Admin
- Linux Security
- Linux Filesystems
- Web Servers
- Graphics & Desktop
- PC Hardware
- Windows
- Problem Solutions

Thinking in C++ Vol 2 - Practical Programming

Prev Home Next

Output string streams

To create an output string stream, you just create an **ostringstream** object, which manages a dynamically sized character buffer to hold whatever you insert. To get the formatted result as a **string** object, you call the **str( )** member function. Here s an example:

```
//: C04:Ostring.cpp {RunByHand}
// Illustrates ostringstream.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
cout << "type an int, a float and a string: ";
int i;
float f;
cin >> i >> f;
cin >> ws; // Throw away white space
string stuff;
getline(cin, stuff); // Get rest of the line
ostringstream os;
os << "integer = " << i << endl;
os << "float = " << f << endl;
os << "string = " << stuff << endl;
string result = os.str();
cout << result << endl;
} ///:~
```

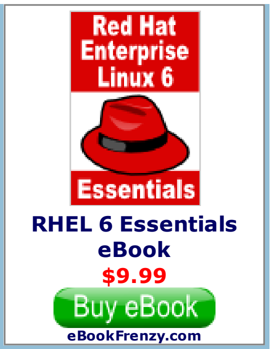
This is similar to the **Istring.cpp** example earlier that fetched an **int** and a **float**. A sample execution follows (the keyboard input is in bold type).

```
type an int, a float and a string: 10 20.5 the end
integer = 10
float = 20.5
string = the end
```

You can see that, like the other output streams, you can use the ordinary formatting tools, such as the **<<** operator and **endl**, to send bytes to the **ostringstream**. The **str( )** function returns a new **string** object every time you call it so the underlying **stringbuf** object owned by the string stream is left undisturbed.

In the previous chapter, we presented a program, **HTMLStripper.cpp**, that removed all HTML tags and special codes from a text file. As promised, here is a more elegant version using string streams.

```
//: C04:HTMLStripper2.cpp {RunByHand}
```



```
//{L} ../C03/ReplaceAll
// Filter to remove html tags and markers.
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../C03/ReplaceAll.h"
#include "../require.h"
using namespace std;

string& stripHTMLTags(string& s) throw(runtime_error) {
size_t leftPos;
while((leftPos = s.find('<')) != string::npos) {
size_t rightPos = s.find('>', leftPos+1);
if(rightPos == string::npos) {
ostringstream msg;
msg << "Incomplete HTML tag starting in position "
<< leftPos;
throw runtime_error(msg.str());
}
s.erase(leftPos, rightPos - leftPos + 1);
}
// Remove all special HTML characters
replaceAll(s, "&lt;", "<");
replaceAll(s, "&gt;", ">");
replaceAll(s, "&amp;", "&");
replaceAll(s, "&nbsp;", " ");
// Etc...
return s;
}

int main(int argc, char* argv[]) {
requireArgs(argc, 1,
"usage: HTMLStripper2 InputFile");
ifstream in(argv[1]);
assure(in, argv[1]);
// Read entire file into string; then strip
ostringstream ss;
ss << in.rdbuf();
try {
string s = ss.str();
cout << stripHTMLTags(s) << endl;
return EXIT_SUCCESS;
} catch(runtime_error& x) {
cout << x.what() << endl;
return EXIT_FAILURE;
}
} ///:~
```

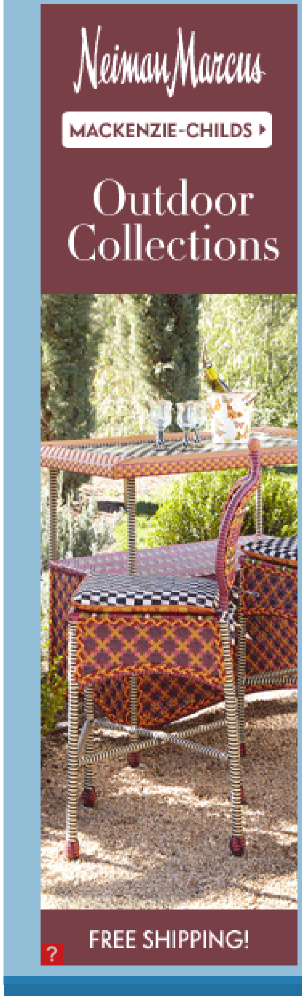
In this program we read the entire file into a string by inserting a **rdbuf( )** call to the file stream into an **ostringstream**. Now it s an easy matter to search for HTML delimiter pairs and erase them without having to worry about crossing line boundaries like we had to with the previous version in Chapter 3.

The following example shows how to use a bidirectional (that is, read/write) string stream:

```
//: C04:StringSeeking.cpp {-bor}{-dmc}
// Reads and writes a string stream.
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
string text = "We will hook no fish";
stringstream ss(text);
ss.seekp(0, ios::end);
ss << " before its time.";
assert(ss.str() ==
```

Adobe
TAKE IT.
MAKE IT.
Adobe Creative Cloud Photography plan.
Lightroom for desktop and mobile plus Photoshop CC.
Just US$9.99/mo.
Join now



```
"We will hook no fish before its time.");
// Change "hook" to "ship"
ss.seekg(8, ios::beg);
string word;
ss >> word;
assert(word == "hook");
ss.seekp(8, ios::beg);
ss << "ship";
// Change "fish" to "code"
ss.seekg(16, ios::beg);
ss >> word;
assert(word == "fish");
ss.seekp(16, ios::beg);
ss << "code";
assert(ss.str() ==
"We will ship no code before its time.");
ss.str("A horse of a different color.");
assert(ss.str() == "A horse of a different color.");
} ///:~
```

As always, to move the put pointer, you call **seekp( )**, and to reposition the get pointer, you call **seekg( )**. Even though we didn t show it with this example, string streams are a little more forgiving than file streams in that you can switch from reading to writing or vice-versa at any time. You don t need to reposition the get or put pointers or flush the stream. This program also illustrates the overload of **str( )** that replaces the stream s underlying **stringbuf** with a new string.