COMP322 - Introduction to C++

Lecture 05 - I/O using the standard library, stl containers, stl algorithms

Dan Pomerantz

School of Computer Science

6 February 2012

Basic I/O in C++

Recall that in C, we have three file handles which are automatically created by the runtime library:

- stdin - standard input (from the keyboard/terminal)
- stdout - standard output (to the terminal/shell window)
- stderr - standard error

C++creates three 'stream' objects which encapsulate these handles:

- cin - standard input (from the keyboard/terminal)
- cout - standard output (to the terminal/shell window)
- cerr - standard error

Formatted output

In C, we use the `fprintf` function (or one of its relatives) to write formatted data:

```
#include <stdio.h>
int n = 101;
float x = 2.1;
fprintf(stdout, "%d %f\n", n, x);
```

In C++, the '<<' operator has been overloaded to write formatted data to `stream` objects:

```
#include <iostream>
using namespace std;
cout << n << " " << x << endl;
```

`<<` `operator`

The `>> operator` is an operator that takes as input 2 operands:

1. Some sort of outstream
2. Some sort of data

It *returns* the outstream, which allows us to use it in another command:

`cout << n << x` is really `(cout << n) << x` . So we first direct `n` to the stream `cout`, then that expression yields `cout`, which allows us to direct `x` to it.

Formatted input

In C, we use the `fscanf` function (or one of its relatives) to read formatted data:

```
#include <stdio.h>
int n;
float x;
fscanf(stdin, "%d %f", &n, &x);
```

In C++, the `'>>'` operator has been overloaded to read formatted data from stream objects.

```
#include <iostream>
using namespace std;
cin >> n >> x;
```

Output stream manipulators

We can control the details of numeric output using stream manipulators, for example:

Name	**Temp**	**Description**
`hex`	N	Print integers in base 16
`oct`	N	Print in base 8
`dec`	N	Print in base 10
`fixed`	N	Set fixed precision
`scientific`	N	Set scientific notation
`left`	Y	Left justify in width
`right`	Y	Right justify in width
`setw(n)`	Y	Set *minimum* field width
`setprecision(m)`	N	Set number of decimal places
`setfill('0')`	N	Set fill character

These manipulators *modify* the stream on the other side of the operator. 'Temp' manipulators apply only to the next item printed.

Using I/O manipulators

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
  int n = 123;
  double x = 123.4567;

  cout << hex << n << endl
       << dec << right << setw(6) << n << endl
       << setfill('*') << left << setw(6) << n << endl
       << scientific << x << endl
       << fixed << setprecision(2) << x << endl;
}
```

Produces the output:

```
7b
   123
123***
1.234567e+02
123.46
```

File input/output

For I/O with named files, we use objects of class 'ofstream' or 'ifstream':

```
#include <fstream>
ofstream out_file;
ifstream in_file;

in_file.open("in.txt");
out_file.open("out.txt");
```

Now one can read or write formatted data from these stream objects:

```
in_file >> n;
out_file << n << endl;
```

When appropriate, we can close the file and reuse the object:

```
in_file.close();
out_file.close();
```

Verifying file operations

The >> operator returns an stream object, which we can treat as a boolean that is true if the operation succeeds:

```
int n;
while (in_file >> n) {
  // process new input
}
```

we can also call member functions to retrieve status:

```
float x;
in_file >> x;
if (in_file.fail()) {
  // operation did not work, 'x' is invalid
}
```

Basic string I/O

You can read (or write) string objects, and write literal strings.
However, an input string is delimited with whitespace:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  char str1[128];  // Character array
  string str2;     // string object
  cout << "Please enter some text:";
  cin >> str1; // No length checking!
  cin >> str2; // May be any length
  cout << str1 << str2 << endl;
}
```

Running this program produces the following example session:

```
bert@gannet$ ./test4
Please enter some text: apple pear peach
applepear
bert@gannet$
```

Reading unformatted characters

If we naïvely attempt to read characters using the >> operator, this will skip spaces, end of line characters, etc.

You can read individual characters, including spaces, using the `get()` method:

```
char ch;
while (in_file.get(ch)) { // Read next character into 'ch'
  // do something
}
```

We can put a character back into the input stream using `unget()`:

```
in_file.unget(ch); // Pretend we didn't read 'ch'
```

Reading line-by-line input

We can read through an entire file using the getline() function:

```
string nextline;
ifstream in_file;
in_file.open("input.txt");
while (getline(in_file, nextline)) {
  // process this line
}
in_file.close();
```

We can then use string functions, or the istringstream class, to parse the line.

Binary files

Many files (e.g. image, sound, database) are binary rather than formatted text.

We also need to implement random access, rather than the sequential access discussed so far.

In C++, we use the `fstream` class to implement this:

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  fstream bstrm;
  const char *filename = "blue.bmp";
  bstrm.open(filename,      // The file name
     ios::in |      // Request read access
     ios::out |     // Request write access
     ios::binary); // Set binary mode

  if (bstrm.fail()) {
    cerr << "Unable to open " << filename << endl;
    return -1;
  }
```

Binary files - reading and writing

We cannot use the << and >> operators to access this file. Instead we use the `get()` and `put()` member functions to read or write unformatted bytes.

We can read individual bytes as follows:

```
  if (bstrm.get() != 'B' || bstrm.get() != 'M') {
    cerr << "Not a bitmap file" << endl;
  }
```

Alternatively, we can read a 32-bit integer from a file like this:

```
int get_i32(fstream& stream, int offset)
{
  int result;
  stream.get((char *) &result, sizeof(int));
  return result;
}
```

Binary files - random access

All streams that are associated with disk files can support random access.

Each stream has a 'get' and 'put' offset which normally begins at zero and advances after each access. This is standard sequential access.

However, the following four functions allow us to implement random access:

- `seekg()` - Set position of next read operation
- `seekp()` - Set position of next write operation
- `tellg()` - Return position get offset
- `tellp()` - Return value of put offset

Binary files - random access

Here is an example of random access - rewriting a particular record in a bitmap file (after C. Horstmann):

```
void invert_pixel(fstream &bstrm) {
  int pos = bstrm.tellg(); // Save current position

   char pixel[3];

   bstrm.get(pixel, sizeof(pixel)); // Read data

   pixel[0] = ~pixel[0]; // Invert
   pixel[1] = ~pixel[1];
   pixel[2] = ~pixel[2];

   bstrm.seekp(pos); // Set write pointer
   bstrm.write(pixel, sizeof(pixel)); // Replace
}
```

Standard Library

The C++compiler comes with a standard library which contains many *very* useful *classes*.

Roughly speaking a class defines a blueprint for an object, which is a complex data type that in addition to storing data can do *actions* on that data.

There are many useful *container classes* that come with C++such as a `vector` class, that works like an array, a `list` class, that works like a linked list, and a `stack` class, that operates like a stack.

One of the goals of *object oriented programming* is to *abstract* much of the detailed logic on how these things work. Rather than having to manage a linked list carefully, we would like to be able to define an *object* that can store all of the information necessary to maintain a linked list, but do so in a way that it's invisible to the user.

Driving a Car

This is similar to how a car works: Originally, to drive a car one had to know a lot about the mechanics of it. Nowadays, some people need to still, but it is not necessary for the average user of a car to know anything other than the various *interfaces* that a car has:

- Pedals to start and stop
- Steering wheel to turn
- Clutch or gear shift of some sort.
- etc.

Template classes

All of these container classes in the standard library are what are called *templated classes*. This means that they are generic and can operate on any data type.

The idea here is to separate the code that has to do with managing the collection from the code that has to do with the data inside the collection.

For example, to insert an element at the end of a linked list, you'll perform the same general code regardless of whether you have an `int` linkedlist or `MatrixElement` linked list. You will go to the end of the list, and then set the last pointer's next to be your new element.

Details

When you use any of these container classes, you have to specify what you want it to contain. You do this using < and >.

For example, to create a vector of int, one would declare a variable.

```
vector<int> numbers;
```

You'll also need to include the necessary header file (e.g. `#include<vector>` or `#include<list>`)

Make sure to include `using namespace std` as well (could also write `std::` each time if you prefer)

Using your object

Once you have created your object, you can use it by calling methods *on* the object. You do this by writing the name of the object/variable/an expression that evaluates to an object of that type, followed by a dot, followed by the method name (and any necessary arguments). For example:

```
vector<int> numbers;
numbers.push_back(3);
numbers.push_back(4);
cout << "The size of the vector is now " << numbers.siz
numbers.at(1) = 3;
numbers.at(100) = 3; //exception! array out of bounds
```

Knowing about the behind the scenes

Although we have abstracted the notion quite a bit, it's still very useful to understand what's going on behind the scenes. This gives you a sense over whether you are using the type correctly or not.

For example, to delete an element from the middle of an array requires shifting every following element over by one. This is not the case with a linked list, however, as you can move the pointers around. This contrast will still be there when comparing vector vs list.

Iterators : Pointers on Steroids

In C++, all of these containers can be *traversed* using what is known as an *iterator*.

An iterator is used like a pointer in many respects, but it has the additional ability to go to the next spot in a container. This means you can do things like add to an iterator.

Suppose I have a `vector<int> foo`, then I can traverse it using the traditional

```
for (int i = 0; i < foo.size(); i++)
```

Behind the scenes, if we had a Linked List, we could traverse it using the `p = p->next` idea.

However, using an iterator, we can combine these ideas.

Iterators from container classes

All of the container classes in stl come with a method to get the beginning of the container and the end of the container. I can access these the same way one calls any other method:

```
list<int> bar;
//insert some elements
.....
// bar.begin() yields the beginning iterator
//bar.end() yields one past the end.
```

Once I store the result of bar.begin() into a variable, I can dereference the variable to get it's data (the pointer aspect of it) or I can increment it using ++ to get the next element (the iterator part of it)

Note, if (*p) gives us a class or struct, we can use dot operator or -> operator normally.

What's up with "one past the end"

When you call the `end()` method on the standard containers, it always returns an iterator that is "one past the end" This can then be used in a comparison to determine if you've reached the end of a container or not. Why does it not just return the end?

The reason is to simplify calculations. The idea is you'll put something into a loop and you can simply write things like `current != container.end()` as a condition.

Additionally, when we see *random access iterators* (in a few slides) you can do things like

```
int size = container.end() - container.begin()
```

This idea is similar to why we write in for loops typically `for (int i=0; i < array.length; i++)` instead of `for (int i=0; i <= array.length - 1; i++)` which is an off by one error waiting to happen.

Type of .begin()

The type of .begin() will depend on the kind of container you have. It will return an iterator to the kind of container you have, but the kind of iterator will depend also on whether the container itself is, for example, read only or not.

For example, if you have a write able `vector<int> foo` then one would write

```
vector<int>::iterator current = foo.begin();
```

On the other hand, if it is a `const vector<int> foo` then the return type of begin() is a const iterator, so you have to write:

```
vector<int>::const_iterator current = foo.begin();
```

Kinds of iterators

There are many kinds of iterators in C++. The type of an iterator depends on the kind of thing one is iterating over. The kind of iterator one has affects what operations they can do on the iterator.

- read only iterators
- write only iterators
- forward only iterators
- bidirectional iterators
- random access iterator

For example, the `begin()` method when called on an array returns a random access iterator. This means you can add arbitrary values to it.

A bidirectional iterator would allow you to use `--` on it.

Example of using iterator

Suppose you have a `list<double>` and you want to print all values of it, one could write a function:

```
printList(const list<double>& elements)
{
     for (list<double>::const_iterator current =
          elements.begin();
          current != elements.end(); current++)
     {
          cout << (*current) << endl;
     }
}
```

Problem with that code

The above code is actually not very general. It only works on `list`. The code would be almost exactly the same if we wanted to use a `vector` instead of a `list`, but currently we'd have to copy and paste.

Typically what we'll do then is write our functions to take as input a *start iterator* and an *end iterator*.

Converting to iterator format

```
printList(list<double>::const_iterator start,
list<double>::const_iterator end)
{
   for (list<double>::const_iterator current =  start;
        current != end; current++)
   {
        cout << (*current) << endl;
   }
}
```

This still has the same problem though that we had to write the word `list` a bunch of times.

Defining our function generically

We can define our function *generically* to take as input a yet to be determined type.

```
template class<Iterator>
print(Iterator start, Iterator end)
{
   for (Iterator current =  start;
   current != end; current++) {
        cout << (*current) << endl;
   }
}
```

At compile time, it will be determined what type `Iterator` actually is. The compiler will check if you are using the iterator correctly. (For example, if you wrote (*current).row it would check if the iterator was over a struct/class with a row property)

Algorithms in standard library

There are many algorithms you can use in the standard library by including `algorithm` in your code. These include the ability to sort, select a subset of a container according to a criteria, count how many elements satisfy a criteria, etc.

Almost all of these functions work on iterators. This means that when you write your own types, these functions can still be used with them.