1 Lab 3

Date: Feb 13, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a # character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to the C++ standard library. After completing this lab, you should have some familiarity with the following topics:

- The use of `std::string`.
- Use of sequence containers like `std::vector`.
- Use of associative containers like `std::map`.

1.2 Exercises

This lab has five exercises. The first two provide motivation for using `std::¬ string` which is used in the third exercise. The fourth exercise sorts numbers using a `std::vector` which is a very popular collection type in C++'s **Standard Template Library** or STL. The final exercise uses vectors, maps and sets to allow querying associations interactively.

Note that this lab provides you an **extremely limited** glimpse into the C++ standard library. Please make sure to check-out the references and sources on the web to get a better idea of what it can do for you.

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab3` branch and a new `submit/lab3` directory. Start a `script` session to log your interaction into a `lab3.LOG` file.

Copy all the lab3 exercises into your `submit/lab3` directory by copying the contents of the `~/cs240/labs/lab3/exercises`:

```
$ cd ~/i240?/submit/lab3
$ cp -r ~/cs240/labs/lab3/exercises .
```

[Unlike the previous lab, we are copying the `exercises` directory rather than its sub-directories as the `exercises` directory contains a `data` directory and a `Makefile` used by multiple exercises.]

Add a `.gitignore` file to the newly created `exercises` directory containing a single line `data`. This will prevent the `exercises/data` directory from being committed to your github repository.

1.2.2 Exercise 1: A Buggy Program

Change over to the `./exercises/1-greet` directory.

```
$ cd exercises/1-greet
$ ls -l
```

You should see that the directory contains a single `greet.cc` file. Read the README to see what the program is supposed to do.

Simply type `make -f ../Makefile` in that directory to build the program. The `-f` option tells it to use a `Makefile` other than that in the current directory; in this case, we are using a `Makefile` from the parent directory.

It should compile a `greet` executable without any errors or warnings. Now try to run it.

```
$ ./greet
usage: ./greet NAME...
$ ./greet bart lisa
```

YMMV, but at different times when I ran this code I got either no output or got the greeting lines printed out with strange characters.

Look at the code and the value which is returned by the `greetMsg()` function. It is clearly a dangling pointer. In fact, it is so obvious that even the compiler spots it and the `#pragma` lines have been inserted to turn off its warning.

We will fix the problem in the next exercise.

1.2.3 Exercise 2: Using new

Change over to the `./2-greet` directory and type `make`. Another `greet` program should be built without any problems.

This `greet` program is very similar to the previous `greet` program except that instead of allocating the memory for the message on the stack as in the previous exercise, the `greetMsg()` allocates the memory for the message on the heap using `new()`.

Run the program for the same test cases as in the previous exercise. It seems to run successfully. But looks can be deceiving, does it really run successfully?

Turns out it is leaking memory as the memory allocated using `new` is never `delete`'d. To verify this, run the program using the memory debugger `valgrind`:

```
$ valgrind ./greet lisa bart
$
```

You should see that it is indeed leaking memory.

Fix the problem by doing a `delete` on the pointer returned by `greetMsg()` when you no longer need it. Since the `new` allocated an array, C++ requires that you use something like `delete[] msg` where `msg` is the return value from `greetMsg()`.

Run the program under valgrind once again to verify that you have indeed fixed the memory leak.

An API like that provided by `greetMsg()` is difficult to use because the caller needs to be sure to free up the memory allocated for the message when done with it. In a large program, it can lead to memory leaks or multiple `delete`'s. In the next exercise, we will using `std::string` from the C++ standard library to solve this problem.

1.2.4 Exercise 3: Using std::string

The C++ language does not support a real string type since a `char *` pointer simply points to an array of bytes. However, the library supports a real string type which manages its own memory and provided all the functionality one would expect from a string type.

Change over to the `./3-greet` directory and look at the code in `greet.cc`. Compile the program as in the previous exercises. Run it using `./greet bart lisa`. It does not run quite right.

Look at the code for `greetMsg()`. It contains the line

```
return GREET + ' ' + std::string(name);
```

`std::string` overloads the `+` operator to perform string concatenation and the above line is an incorrect attempt to use it to concatenate `GREET`, a single space character and the `name` parameter.

Since `+` associates to the left, the above line is equivalent to

```
return (GREET + ' ') + std::string(name);
```

Consider what the compiler sees as the operands for the first `+`: `GREET` which is a `char *` and `' '` which is a `char` which is an integral type. So it interprets the first `+` as adding an integer to a `char *` pointer, resulting in a `char *` pointer which is invalid.

Now when the compiler sees the second `+`, the two operands are the invalid `char *` pointer and a `std::string`. Presumably `std::string` has set things up so that addition of a `char *` with a `std::string` results in concatenation. However, in this case, the invalid `char *` pointer is pointing to some memory which gets treated as a C string by the overloaded `+`. This explains the strange output you may have seen.

The fix is to set things up so that the first `+` can match the overloaded `std::string +` operator. This can be done by simply converting its first operand to a `std::string`.

1. Make the necessary changes and run the code. Verify that it is now working correctly.

2. Run the program under valgrind and verify that it is not leaking memory.

3. Print out the size of the return value from `greetMsg()` using `sizeof()`. Provide really long names to the program, you should see that the size of the return value does not change.

The fact that the size of the return value does not change means that the `std::string` return value is simply some kind of header with the actual string content stored elsewhere, namely on the heap. The fact that the program does not leak memory even though there are no explicit calls to `delete` means that `std::string` must be using some kind of smart pointer behind the scenes to ensure that all memory is freed up without any involvement by the user of `std::string`.

1.2.5 Exercise 4: Using std::vector

Change over to the `./exercises/4-sortnums` directory and look at the code in `sortnums.cc`. It is set up to read `int`'s from the files specified by its command-line arguments into a dynamically grown `std::vector`, sort that vector and write out the results of the sorted vector on standard output.

1. The reading of the `int`'s is handled by `readNums()`. It's operation should be quite clear from the code; it uses the `>> istream` operator to read the next `int` from `istream in` and adds it the the vector using its `push_back()` operation.

2. Once `readNums()` returns to `main()`, `main()` sorts the returned vector using the `sort()` function provided by the `algorithm` header file. To understand its operation, please look at its documentation. You will see

multiple overloaded `sort()` headers for different versions of C++. Basically, `sort()` takes two "pointers" pointing to the start of the sequence to be sorted and one beyond the end of the sequence. That sequence should allow "random iteration" meaning it is easy to jump back-and-forth within the sequence. So the code in `main()` sorts the `nums` vector by specifying the start of the iterator as `nums.begin()` and the end of the iterator as `nums.end()`.

[Many STL containers provide similar `begin()` and `end()` iterators.]

No comparison function is provided to `sort()` in our code. So `sort()` will simply use the default ordering of the data type stored in the vector. In our case, we are storing `int`'s, so the ordering used is `<=` on `int`'s.

3. The `sort()` is followed by a single line loop which outputs the `int`'s from the sorted `nums` on to standard output.

Compile and run the code. There are a couple of small data files in this directory:

```
$ sortnums *.dat
```

Modify the provided code so as to change the `sort()` to sort in non-ascending order. According to the documentation, you will need to provide a comparison function as the third argument to the `sort()` function. This comparison function `comp()` should return true if its first argument is less than (i.e. ordered *before*) the second. Its signature should be:

```
bool cmp(const int &a, const int &b);
```

Write this comparison function, and retest to verify that your output now uses the reverse order.

1.2.6 Exercise 5: Mapping Numbers to Paths

Change over to the `./exercises/5-numpaths` directory. The program in file `numpaths.cc` reads in `int`'s from the files specified by one-or-more command-line arguments. It sets up an association between each number and the file(s) which contain it. It then enters an interactive loop where it reads an `int` from the input and outputs the names of all the files which contain it.

Compile and run the program:

```
$ make -f ../Makefile
$ numpaths ../data/*.dat
>> 342
```

```
342: ../data/4.dat ../data/8.dat
>> 324
324: ../data/1.dat ../data/3.dat ../data/8.dat
>> 45
45: NOT FOUND
>> 234
234: ../data/6.dat ../data/8.dat
>> #type ^D for EOF
$
```

We essentially need a mapping between each `int` and the collection of filenames of the files which contain that `int`. Since the same `int` can occur in the same file multiple times, we would like to avoid duplicate filenames. We can avoid duplicates by using a `set` to represent the collection of filenames.

So our basic data-structure will be a map from `int`'s to a set of strings representing filenames. In C++ template notation, we write this map as `map<int, set<string>>`.

A few points are worth noting:

- When filling in the map in the `readNums()` function, we do so using its overloaded `[]` operator. This has the happy side-effect that if do a lookup of `map[n]` when `n` has not been seen earlier, then the `map` will create a new slot in the map initialized to an empty set and return a **reference** to the new set. Hence the `insert()` on the returned set will add in the corresponding filename.
- When we are querying the map in the interactive loop, we do not want a query for a non-existent `int` to create a new slot in the map. So we do not use the `[]` operator; instead we use the `map`'s `at()` member function which throws an exception when the `int` does not exist.

Having read and understood the program, modify it so that instead of mapping `int`'s to filenames, it maps each filename to the sum of the `int`'s in that file. Change the interactive loop to allow the user to type in a filename (which in general will be a path). If the filename is known to your program, it should print out the sum of all the `int`'s in that file.

1.2.7 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab3.LOG` in your `lab3` directory. Add all your files to git and commit. Then merge your `lab3` branch into the `master` branch and commit your changes.

1.3 References

Online *<https://en.cppreference.com/w/>*.

Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition, Addison-Wesley, 2012.

Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, 2001.