# Standard Template Library (STL)

During this tutorial, we’ll try out some of the STL stuff. For this we’ll make use of the concept of a App manager that stores metadata apps. As you work through this tutorial sheet you will need to include several libraries that give access to the STL features. I have given hints about which libraries you need, but if you attempt something that isn’t recognised, make sure you have the required libraries by checking the MSDN documentation (the F1 key)

1) Create a new Empty project and add a new main method

2) Download the MyDate.h, AppMetadata.h and AppMetadata.cpp files from the Blackboard site and include them in your Visual Studio project (copy the files into the project folder before adding them into the project).

3) Create a linked list to store AppMetadata objects (you will need to include the “list” header file)

4) Add in half a dozen or so apps (perhaps look at your phones to get some ideas, although any data is sufficient for today). An example of adding data into a list collection called lapps is given below:

lapps.push\_back(AppMetadata("BBC News", 1000, MyDate(1, 1, 2016), MyDate(30, 10, 2017)));

5a) Write some code that iterates through the collection and displays the apps to the screen (you will need to include the “iterator” header file) using an iterator

5b) You can also perform the output display using the for\_each algorithm (include the algorithm library).

for\_each(lapps.begin(), lapps.end(), displayApp);

Where displayApp is a global method that takes a const reference to an AppMetadata object and displays it to the screen. Write a new bit of code that displays all the data in the collection using this technique.

6) What do you notice about the created variables inside these objects? Perhaps drop a breakpoint into your code that allows you to view the objects in the list using the debugger.

7) Write some code that uses an iterator and the copy algorithm to copy the data from the list into a vector (you will need to include the “vector” and the “algorithm” header files). You will need an insert iterator adapter.

8) What do you notice about the created variables inside the AppMetadata objects that exist in the system? What is going on? Is this what you would expect?

9) Sort the vector collection on the app name. The following method signature might be useful:

bool sortOnName(const AppMetadata& lhs, const AppMetadata& rhs)

You will need to implement the body, but it should return true if lhs comes before rhs based on the name of the app.

You can then sort it using the following code (where vapps is my vector collection):

sort(vapps.begin(), vapps.end(), sortOnName);

10) What happens when you try and sort the list using a modified version of the above code?

11) What do you notice about the created variables inside the AppMetadata objects that exist in the system before and after the sort method is called? What is going on?

12) OK, enough with the asking “what is happening with the create variables” question… you should have spotted that it keeps increasing way more than it should. This is because we are using “in-place” objects on the stack that are constantly being shifted from one block of memory to another. This is inefficient – each time a AppMetadata object is moved or copied, there is a non-trivial amount of data being moved about. The use of heap objects would alleviate this problem as only memory pointers are copied (4 bytes for 32bit and 8 bytes for 64bit).

13) Before updating your code to work with heap memory, let’s look at those bind1st and bind2nd function adapters.

We could use a line of code similar to:

bool isNameEqual(const AppMetadata& lhs)

{

return lhs.getAppName()=="Mail";

}

list<AppMetadata>::iterator res=find\_if(lapps.begin(), lapps.end(), isNameEqual);

to find items in a collection that match a specific name. However, it is rather limited because we have hardcoded our app name to “Mail”. A more generic approach would be to use a method like:

bool isNameEqual(const AppMetadata& lhs, const string search)

{

return lhs.getAppName()==search;

}

To call this method as part of the find\_if algorithm, we need to bind the dynamic search parameter (i.e. the string search parameter). The third parameter of the algorithm will thus be:

bind(isNameEqual, \_1, "Mail")

The \_1 placeholder is part of the std::placeholder namespace and will need to be used. For functions that have more parameters, there are additional placeholders: \_1, \_2, \_3, etc… You could also use lambda calculus to write an in-place function.

The following section of code uses bind2nd, which is deprecated in C++11 and removed in C++17. This has been kept in for interest as you may still see existing code that uses these adapters.

Using bind2nd to bind a value to the search parameter: You will need to wrap the isNameEqual function into a class and move the body of code into the operator() method:

class TestNameOnAppMetadata : public binary\_function<AppMetadata, const string, bool>

{

public:

bool operator() (AppMetadata lhs, string search) const

{

return lhs.getAppName()==search;

}

};

Notice that the arguments between the < and > symbols on binary\_function are template data types and should be the data types of the first parameter, second parameter and return type respectively.

You could also use the correct template parameter matching constructs:

class TestNameOnAppMetadata : public binary\_function<AppMetadata, const string, bool>

{

public:

result\_type operator() (first\_argument\_type lhs, second\_argument\_type search) const

{

return lhs.getAppName()==search;

}

};

At compile time, result\_type, first\_argument\_type and second\_argument\_type will be mapped according to the template data types (bool, AppMetadata and const string in this case).

You can then use this method in your find\_if algorithm using the bind2nd function adapter as follows:

list<AppMetadata>::iterator res=find\_if(lapps.begin(), lapps.end(), bind2nd(TestNameOnAppMetadata(), "Mail"));

Give it a go… (there is no line break in the above code, it has just word-wrapped)

14) Update your code to work with heap memory objects. Your data type for the list and vectors would now be an AppMetadata pointer:

list<AppMetadata\*> lapps;