Project 10: Sorting Demo

This programming project has several purposes:

1. Gain experience with various well-known sorting algorithms.
2. Understand inheritance, interfaces, and virtual functions.
3. Understand the factory pattern.
4. Understand pointer arithmetic.

Before I get into how to complete this project, I’m going to go summarize a few of the concepts that I went over in class. I’ll talk about interfaces and the factory pattern. Then I’ll talk about the basics of pointer arithmetic, which you see in the algorithms in the slide deck. After that, I’ll walk through how you would go about creating implementations for the various sorting algorithms required for this project.

**If you don’t need a review of what I covered in class, you can skip straight to section 4.**

# What is an interface?

An interface is a class that has only public pure virtual method definitions. There’s a lot of words here, so let’s break it down:

1. **Public:** this means the method can be called from outside the class.
2. **Virtual method:** A virtual method is a method that is defined in a base class and given a default body—however, an inheriting class can override (i.e., essentially replace) that method with their own implementation. You define a virtual method in your base class by putting the keyword, virtual, in front of its definition.
3. **Pure virtual method:** A pure virtual method is a virtual method defined in a base class that has no body. A pure virtual method is made by putting “= 0” at the end of the definition in place of the body.

It is good practice to begin all interfaces with the capital letter I. For this project, you will use the ISortingAlgorithm interface.

Here is a listing of the ISortAlgorithm interface. It has only one method.

class ISortAlgorithm

{

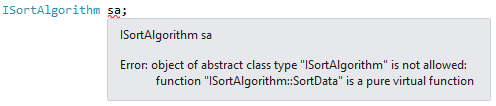
public:

virtual void SortData(int \*data, size\_t count) = 0;

};

Notice the parts of this declaration that I’ve highlighted.

You cannot make an instance of this class. If you do, you’ll get a compiler error saying that “SortData is a pure virtual function. This method has no body.”



## Interface Example

The SortData method MUST be overridden by a deriving class, and the deriving class must provide a body. Let’s do that now. We’ll define a class called BubbleSort. Here is what BubbleSort.h should look like:

#pragma once

#include "SortFactory.h"

class BubbleSort :

public ISortAlgorithm

{

public:

BubbleSort();

~BubbleSort();

void SortData(int \*data, size\_t count);

};

I’ve highlighted the definition for the SortData method. Since this is a header file, we won’t provide a body. That has to be done in the .cpp file.

Here is a listing of BubbleSort.cpp. Notice how you have to put a reference to the BubbleSort class in front of the method name. This tells the compiler that the method belongs to the BubbleSort class.

#include "BubbleSort.h"

void BubbleSort::SortData(int \*data, size\_t count)

{

bool sorted;

for (size\_t n = count; n >= 1; n--)

{

sorted = true;

for (size\_t i = 1; i < n; i++)

{

if (data[i - 1] > data[i])

{

sorted = false;

Swap(data[i], data[i - 1]);

}

}

if (sorted)

break;

}

}

And that’s it. You’ve created a new sorting class, and the class properly implements the ISortAlgorithm interface.

## Ok, interfaces, so what?

So why would you do this sort of thing?

Well, that’s what this lab is about. We want to have classes to encapsulate various sorting algorithms, but we only want to manage them through the ISortAlgorithm interface. This way, I can have a pointer (or a reference) to an ISortAlgorithm interface, and I don’t care what sort algorithm I’m using, I just want my data sorted. I can sort my data through the interface.

This will be clearer with an example.

Suppose we have a function that creates all of our various sorting classes. We tell it what kind of sorting class to instantiate (say using an enumerated type), and it allocates a new instance of the class, and returns just the ISortAlgorithm interface to that class.

Here’s what I’m talking about:

enum SortType

{

SortType\_BubbleSort,

SortType\_SelectionSort,

SortType\_InsertionSort,

SortType\_ShellSort,

SortType\_HeapSort,

SortType\_MergeSort,

SortType\_QuickSort

};

ISortAlgorithm \*SortAlgorithmMaker(SortType type)

{

switch (type)

{

case SortType\_BubbleSort:

return new BubbleSort();

case SortType\_SelectionSort:

return new SelectionSort();

case SortType\_InsertionSort:

return new InsertionSort();

case SortType\_ShellSort:

return new ShellSort();

case SortType\_HeapSort:

return new HeapSort();

case SortType\_MergeSort:

return new MergeSort();

case SortType\_QuickSort:

return new QuickSort();

}

// Unknown type!

return NULL;

}

Now we can pick and choose which sorting method we think is most appropriate, have one made for us, and then call it through the ISortAlgorithm interface.

Here is an example function that sorts an array of integers of arbitrary size using one of our sorting classes:

void SortSomeData(int \*data, size\_t count, SortType type)

{

// First, create a new sort class, and get its interface:

ISortAlgorithm \*sorter = SortAlgorithmMaker(type);

// Now sort our data

sorter->SortData(data, count);

// All done. Delete the sort class.

delete sorter;

}

Using this approach, we can choose whichever sorting strategy best suits our needs. The ISortAlgorithm interface acts as a façade, abstracting away the details of what is actually going on behind the scenes. Thus, we can focus on what we need to do, and not worry about how it gets done.

The pattern of using interfaces is much more powerful than what you see here. They are the mechanism by which object oriented libraries expose their services to other applications.

# The factory pattern

The factory pattern is a design pattern where we have two or more classes. The first class is a factory. Its purpose is to build things. What it builds is instances of classes, all of which inherit a common interface.

This project uses a simplified version of the factory pattern. If you look in SortFactory.h, you will see a declaration for the SortFactory class, which creates instances of the various sorting classes:

class SortFactory

{

public:

static ISortAlgorithm \*GetSortAlgorithm(SortType);

};

This class only has one method, which is static. A static method is a method that does not belong to an instance of a class; in other words, you do not have to make an instance of the SortFactory class to call the GetSortAlgorithm() method. This can be an advantage in some situations, which I’ll demonstrate shortly. For the most part, a static method behaves like a function in C.

In the source code that I provided for this project, you can see the implementation for the GetSortAlgorithm method:

ISortAlgorithm \*SortFactory::GetSortAlgorithm(SortType sortType)

{

// TODO:

// Add cases to this switch statement to return a new instance of each

// sort class.

switch (sortType)

{

case SortType\_BubbleSort:

return new BubbleSort();

}

throw exception("Sort type is not implemented.");

}

It’s pretty straightforward. You will have to expand this method to return more sorting classes besides BubbleSort (more on that later).

When calling a static method, you need to put the class name in front of the method, and separate the two using a double colon (the double colon “::” is called the scope resolution operator).

Here is how we would create a new instance of the BubbleSort class, and get its ISortAlborithm interface:

void SortByBubbleSort(int \*data, size\_t count)

{

// Create an instance of the BubbleSort class, and obtain its

// interface.

ISortAlgorithm \*sorter =

SortFactory::GetSortAlgorithm(SortType\_BubbleSort);

// Sort the data

sorter->SortData(data, count);

// De-allocate the BubbleSort sort class.

delete sorter;

}

So, what’s the point of doing it this way, as opposed to the way I just showed you earlier (using a C function, instead)?

This way is more object-oriented. Take a look at main.cpp. Notice how little code I had to write, in order to test all seven sorting algorithms? I have just one test case, and I call it seven times with a different algorithm type.

# Basics of pointer arithmetic.

I want to touch briefly on some pointer tricks I use in the slide deck and throughout this project. Do not worry if this part becomes confusing. You can skip this section, and go on to section 4 if you like, but I want to go over this because if you can understand pointer arithmetic, pointers will never again be a mystery to you.

Pointer arithmetic can be summed up in one simple idea: the compiler sees no difference between a pointer to a given type and an array of that same type.

What on earth am I talking about?

Let’s take a look at the declaration of BubbleSort’s SortData() method:

void SortData(int \*data, size\_t count);

Notice how the data parameter is declared? It’s just a pointer. It’s not an array.

Hang on, you may ask—if it’s a pointer, how can it point to an array? All it points to is one integer, it doesn’t point to an array of integers!

Well, it turns out that C (and by extension C++) lets us treat ANY pointer as an array.

So, that means we can do stuff like this:

cout << data[32];

And what happens if the array doesn’t have 33 items? Well then you’re in trouble, aren’t you? That’s why we also passed in a count of items—and we need to make sure that we don’t exceed that count.

So, how do we pass in an array to the SortData method? Do we need any kind of special syntax, like putting a & in front of it?

Nope. Just pass it in, like you would any other variable. If it’s an array, it’s ALREADY a pointer.

In this example code, both the SetArray function and the SortData method take a int\* as their first parameter (you can see the definition for SetArray at the top of main.cpp):

// Declare our data array, and populate it with random stuff.

int data[100];

SetArray(data, 100);

ISortAlgorithm \*alg = SortFactory::GetSortAlgorithm(sortType);

alg->SortData(data, 100);

As a side-note, and to further illustrate, suppose we have a real int\*, and we assign it to a real int (not an array of ints), like so:

int a = 100;

int \*p = &a;

It turns out that we can treat p like an array with one element. The following code will set the value of variable a to 10, using p:

data[5] = 10;

cout << a; // this will print 10

Are you thoroughly confused?

I hope not. If you are, remember that you can skip this section and go on. Just don’t try and set p[1] to something, or bad stuff will happen. ☺

I want to point out one more thing, and then I’ll move on.

You can add an offset to any pointer, like so:

data + 5

This will yield a pointer to the element in the array at position 5. Whenever you add a value to a pointer, you move the pointer over by the size of its type. So if data is declared as an int\*, and an integer takes up four bytes, then “data + 5” will move the pointer over by 20 bytes.

Whenever you add a value to a pointer, you get a new pointer. You can dereference this pointer and get the value stored at that address, or set the value—anything you could normally do with a pointer. The only thing you have to remember is to put parentheses around the addition.

Here is how we would set the value in the array at index 5 using pointer arithmetic:

\*(data + 5) = 10;

This is the same thing as doing this:

data[5] = 10;

So, why would you ever use \*(data + 5)? Well you wouldn’t, but I DO use “data + n” in the slide decks for quicksort and merge-sort, because that’s the easiest way to get a pointer to a sub-portion of an array.

You’re probably wondering why we have to use parenthesis. Well, that’s because it turns out that the \* in C has a higher precedence than the +. In other words, in this line of code:

\*data + 5 = 10;

The computer will evaluate \*data (which de-references data and gets the value it was pointing to), and then add 5, and then try to assign 10 to that…which is quite meaningless.

Anyway, if you’re really curious about this and you need to talk more about it, see me outside of class.

Onward!

# How to complete this assignment

You will complete this assignment by creating sorting classes for each of the following sorting algorithms:

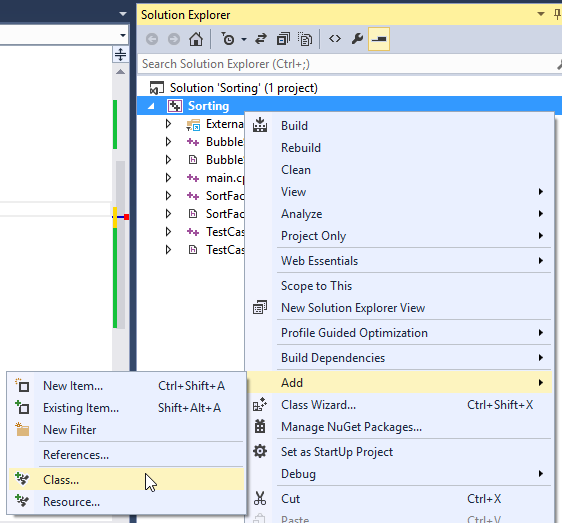
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort
5. Heap Sort
6. Merge Sort
7. Quicksort

I’ve created an example BubbleSort.h and BubbleSort.cpp for you so that you can get the hang of things. You will still have to provide an implementation for BubbleSort’s SortData() method.

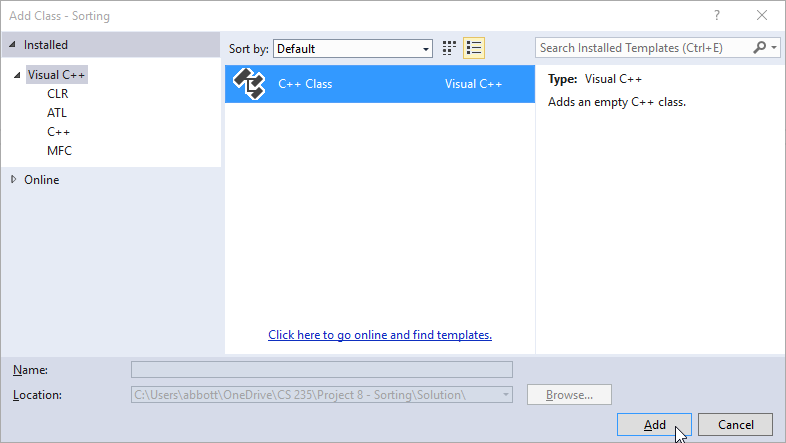
So, let’s walk through how to do this. Let’s do SelectionSort.

## Add a new class

In visual studio, the easiest way to create a new class is to right-click on the project, then from the context menu select: Add🡪Class…

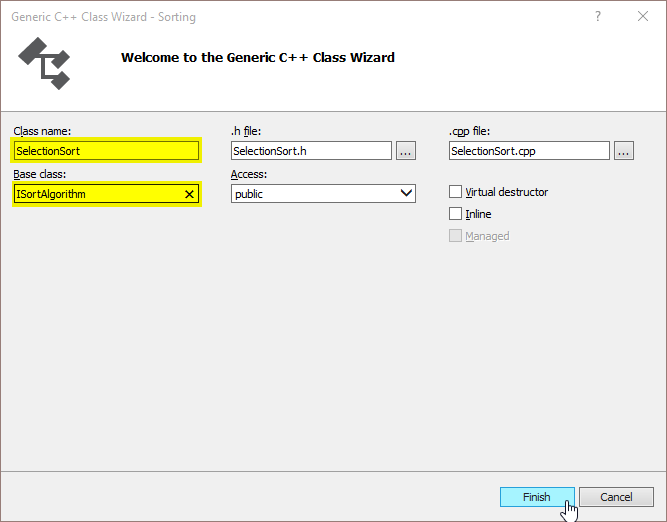


A dialog box will appear.

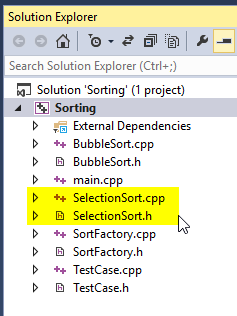


Make sure “C++ class” is selected, then click the Add button.

Another dialog will appear. Just fill out the parts I highlighted in yellow. Visual Studio will fill out the rest of the fields:



Click the Finish button when you’re done, and you should have a new .h and .cpp file in your project:



## Implement the ISortAlgorithm interface

Now we need our new class to implement the ISortAlgorithm interface.

Open up SelectionSort.h, and add a declaration for the SortData method. This declaration must match the declaration found in the ISortAlgorithm interface exactly (minus the virtual keyword, and the “=0” at the end). Do it just like you see in BubbleSort.h.

#pragma once

#include "SortFactory.h"

class SelectionSort :

public ISortAlgorithm

{

public:

SelectionSort();

~SelectionSort();

void SortData(int \*data, size\_t count);

};

Now open up SelectionSort.cpp. Go to the end of the file, and declare the implementation for the SortData method. You’ll have to get the actual code from the slide deck.

void SelectionSort::SortData(int \*data, size\_t count)

{

// TODO: go to the slide deck, and type the algorithm

// in here.

}

And that’s it. Easy-peasy.

## Modify SortFactory.cpp

Now you’ll have to add a case statement for the SelectionSort class to the factory. Go to SortFactory.cpp.

At the top of the file you’ll see a TODO comment. You’ll need to add your new header file there.

// TODO:

// Add include files for each of your sorting classes here:

#include "BubbleSort.h"

#include "SelectionSort.h

Next, find the case statement in GetSortAlgorithm, and add a new entry. Do it just like with BubbleSort:

ISortAlgorithm \*SortFactory::GetSortAlgorithm(SortType sortType)

{

// TODO:

// Add cases to this switch statement to return a new instance of each

// sort class.

switch (sortType)

{

case SortType\_BubbleSort:

return new BubbleSort();

case SortType\_SelectionSort:

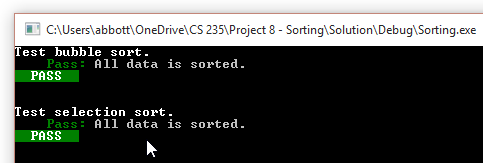
return new SelectionSort();

}

throw exception("Sort type is not implemented.");

}

And that should do the trick. You should be able to run the tests, and the test case for selection sort should pass.



Bravo!

Now go and do it for all the other sorting classes, and don’t forget to provide an implementation for BubbleSort.

# Submitting your assignment

You will need to zip up everything for this assignment. I want to see all your code.

Before you upload your .zip file to learningsuite, please open it up (or unpack it into a temporary directory), and make sure it has all your files in it. I get a lot of .zip files that contain just the .sln file.

Good luck!