Using an IDE

These must be completed and shown to your lab TA either by the end of this lab, or at the start of your next lab. You may work in groups of up to two people.

In this lab you are going to create a C++ class that contains an array and operations on that array. This lab will guide you through the process of creating the class in Microsoft Visual Studio. Other IDEs may function differently. You are not intended to come up with the program code yourself, but please copy and paste each section of the program into a text editor and compile and run it. The code itself should be a review of object-oriented programming with C++.

Writing a C++ Class

.h and .cpp Files

C++ classes are made up of a header file and an implementation file. Both files should have the same name except that the header file has a .h extension while the implementation has a .cpp extension. The header file contains the class interface, and the .cpp file contains the implementation. The header file consists of the class name, and the name (and type) of the member variables and the header for each of the methods. The .cpp file consists of the definition for each of the class methods.

Public or Private?

Class member variables and methods should be specified as being either *private* or *public*. Private variables or methods can only be accessed from within the class, whereas public variables and methods can be accessed from outside the class. There are a couple of good general design principles to follow when deciding whether or not to make a method or variable public:

- Only make something public if it needs to be public; note that not all methods need to be public, many classes have private helper methods that never have to be accessed directly from outside the class.
- Make all member variables private, if necessary write public getter and setter methods for the variables. This allows you to write the setter methods to ensure that any class invariants are maintained (e.g. such as ensuring that an array size is a positive number). There are exceptions to this principle but you won't encounter any in this example.

Constructors and Destructors

Every class requires at least one constructor that creates new objects of that class. A class will often have multiple constructors that build new objects in slightly different ways. A constructor is a method that has *exactly the same name as the class and has no return type*; it is responsible for setting the initial values of the member variables of an object.

C++ classes also require destructors. A destructor is responsible for de-allocating any dynamic memory that an object uses. If you do not write a destructor for a class a default one is created for you. This class will allocate dynamic memory for the array so will require a destructor.

Class Description

The class creates an array in dynamic memory of a given size. Users are allowed to insert values in the next available space in the array and to change existing values, but are not allowed to insert values in such a way that there are "gaps" of unused elements in the array. Users are allowed to retrieve values from the occupied section of the array. Users are not allowed to delete values (although, as described above, they may change them).

The class will have the following *public* methods:

- default constructor – allocates space in dynamic memory for an integer array of size 10
- constructor(int) – allocates space in dynamic memory for an integer array of the given size
- destructor
- Insert(int) – sets the value of the next free element of the array to the parameter
- Get(int) – returns the value of the array element at the given index
- Set(int, int) – sets the value of the array element at the first parameter's index to the second parameter's value
- MaxSize – returns the size of the underlying array
- Size – returns the number of values currently stored in the array

The class should also have the following *private* attributes:

- a pointer to an int (that will refer to the array)
- an integer that records the actual size of the array
- an integer that records the number of values stored in the array

Writing the Header File

The header (arrayclass.h) file should contain the class definition, which should be separated into public and private sections. The completed header file may be downloaded from the course website. From a new Visual C++ empty project, right click on the Header Files in the Solution Explorer pane, and either select Add->New Item, create a new .h file and paste the contents, or choose Add->Existing Item and locate the saved arrayclass.h file.

Header File Notes

- PRE stands for pre-condition - something that must be true for the method to function correctly; POST stands for post-condition – the state of the program after the method has run; PARAM stands for parameter
- All the methods are public because they might be used (called) by modules or functions outside the class, for example a function might want to know the size or contents of an `ArrayClass` object
- All the attributes are private so that their values can be protected from inappropriate changes by non-class functions, for example changing the `current_size` of the array without actually adding a new value to the array

Writing the Implementation File

The implementation file (arrayclass.cpp) should contain the definitions (i.e. implementations) of each of the class methods.

Writing the .cpp file entails writing each of the methods one by one. In practice this usually involves writing a constructor and one or two methods then testing them rigorously before moving onto the next method. We won't follow this process in the lab, partly because the class is fairly simple and partly to save time. Completed function bodies are provided below which you may paste into your class .cpp file; please try to understand the syntax and operation of each before moving on.

The .cpp should include the header file and each method should be given its fully qualified name, which includes the name of the class. Both of these are illustrated below.

Constructors
Here is the first part of the .cpp file which contains the default constructor:

```
// Default constructor
// POST:  Creates an ArrayClass object with an array of size 10
// PARAM:
ArrayClass::ArrayClass()
{
  arr_size = 10;
  arr = new int[arr_size];
  current_size = 0;
}
```

- A constructor initializes the attributes of an object, for this class this entails setting the size of the array, creating the array in dynamic memory and setting the number of elements currently stored in the array to zero
- The variables `arr_size` and `current_size` are attributes of an `ArrayClass` object, and are not variables declared in the constructor; changing the first line to `int arr_size = 10` would mean that `arr_size` was a local variable belonging to the constructor and the object's `arr_size` attribute would remain un-initialized – which would be problematic
- Constructors have the same name as the name of the class and unlike other methods (and functions) do not have a return type
- `ArrayClass::` is the name of the class to which the constructor belongs and tells the compiler that this method is the implementation of the `ArrayClass` default constructor; every method's name must be preceded by the class name and the scope resolution operator (`::`)

The second constructor is almost identical except that `arr_size` is set to the value of the parameter. It is common for classes to have multiple constructors whose prototypes (the method header) differ only by their parameter lists.

```
// Parameterized constructor
// PRE:   array_size > 0
// POST:  Creates an ArrayClass object with an array of size array_size
// PARAM: array_size = size of the array to be created
ArrayClass::ArrayClass(int array_size)
{
  int arr_size = array_size;
  arr = new int[arr_size];
  current_size = 0;
}
```

Destructor

The destructor is responsible for de-allocating any dynamic memory that has been allocated for an object. The name of the destructor is always the name of the class preceded by a tilde (~), like constructors destructors do not have return types. Destructors should never (or at least very rarely) be called explicitly. They are invoked either explicitly by a call to `delete` or automatically when an object's lifetime has expired.

```
// Destructor
// POST:  De-allocates dynamic memory associated with object
ArrayClass::~ArrayClass()
{
  delete[] arr;
}
```

- The `delete` command deletes any dynamic memory associated with a pointer; in this case it must be followed by `[]`s since `arr` points to an array

Insert

The Insert method should set the value of the next free element to its parameter.

```
// Sets the value of the next free element
// PRE:   current_size < arr_size
// POST:  sets index current_size to value
// PARAM: value = value to be set
void ArrayClass::Insert(int value)
{
  if (current_size < arr_size)
  {
    arr[current_size++] = value;
  }
  // NOTE – no else – should throw error
}
```

- The method sets the appropriate element of the array and increments `current_size`; making `current_size` private ensures that its value is always consistent with the state of the object, if it was public it could be directly accessed by a programmer and given an inappropriate value
- The method ensures that values are not inserted at invalid indices, preventing users from attempting to insert a value in an array that is full

Set

The Set method sets the value of an element whose index is passed to the method.

```
// Sets an existing element's value
// PRE:   0 <= i < current_size
// POST:  sets index i to value
// PARAM: i = index of element to be changed
//        value = value to be set
void ArrayClass::Set(int i, int value)
{
  if (i >= 0 && i < current_size)
  {
    arr[i] = value;
  }
  // NOTE – no else – should throw error
}
```

- This method changes an existing value, so does nothing if the index is invalid

Get

The Get method returns the value at a given index

```
// Returns an element's value
// PRE:   0 <= i < current_size
// POST:  returns the value at index i
// PARAM: i = index of value to be returned
int ArrayClass::Get(int i) const
{
  if (i >= 0 && i < current_size)
  {
    return arr[i];
  }
  // Returns 0 if i invalid – THIS IS NOT SATISFACTORY!
  else
  {
    return 0;
  }
}
```

- In the previous methods the approach to dealing with incorrect input was to do nothing when a method is given an index that is invalid (when it would fall outside the array, or because it does not refer to an existing element); this approach does not work for the get method since the method must return a value
- The approach taken in this example is to return zero; it is important to understand that *this is not a satisfactory solution* to the problem since an existing array element could contain the value zero
- This issue is discussed briefly under Error Handling at the end of the document

MaxSize and Size

These functions return the size of the underlying array and the number of items currently stored in that array.

```
// POST:  Returns the size of the underlying array
int ArrayClass::MaxSize() const
{
  return arr_size;
}

// POST:  Returns the number of elements stored in the array
int ArrayClass::Size() const
{
  return current_size;
}
```

Testing the Class

Add the test suite .cpp file containing the main function to your project sources and compile/run it. This is a very quick and dirty test of your class, kept to a minimum for the sake of time. In practice each method should be rigorously tested as it is implemented before moving on to the next method. This is particularly important when one method relies on another. For example I would suggest never starting to implement a method that removes items from a data structure before you are absolutely certain that your insertion method works perfectly.

With the copy/pasted code, you should have the following output:

```
7 6 5 13 3 2 1
ac1: current size = 7, max size = 10
ac2: current size = 0, max size = 0
```

Items were not successfully added to ac2, why? Place a breakpoint next to the ac2.Insert(2) call in the test driver .cpp file. Run your program, and it will pause at this breakpoint. “Step Into” the Insert call, and once there, use “Step Over” to execute one line at a time once your execution is inside the Insert function. The if statement gets skipped, even though we expect that the value of current_size should satisfy the condition.

Inspect the local variables window. You should see that current_size is 0, but expanding “this” you can see the member attributes of the calling object. arr_size is 0. But we are positive we initialized this value correctly in the parameterized constructor, or did we really? Identify the error, fix and recompile and run.

Error Handling

One of the advantages of using classes is that it allows programmers to relatively easily ensure that their object attributes are never given inappropriate values, and to identify input values that are invalid. This class does recognize input errors and, as much as possible, protects the class from these errors; however the existence of errors is not conveyed to a programmer using the class.

A more sensible approach to the one taken in the implementation would be to make the methods throw an exception if their input data was invalid. Add appropriate error handling to the Set and Get functions as shown in class.

Templates

The ArrayClass class is currently written to store integers. Following the example shown in class, modify the ArrayClass to accept and store in its dynamic array, a generic template type. The test driver file will also be modified to create ArrayClass objects with a template type.