4.2a - Introduction to Templates

Exercise 1: Templated Array Class

In a previous exercise we created an *Array* class that stores *Point* objects. The disadvantage of this class was that it only stores *Point* objects. If we want to have an array class for *Lines*, we need to copy the code and replace references to *Point*to *Line*. This is a lot of work and makes maintenance more difficult. A bug in one version has to be fixed in the other versions as well. Better is to use the same code for different data types. This is possible using templates as shown in Figure 1.

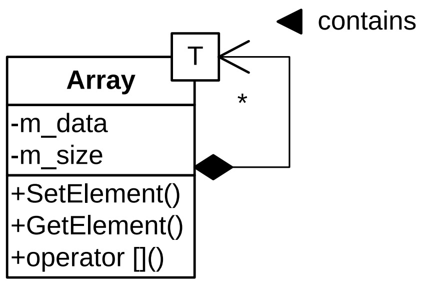

Figure 1: Templated Array class containing Ts

Thus transform the *Array* class for points created earlier into a template class:

- In the header file, declare the *Array* class as a template of type *T*.
- Replace all references to *Point* with *T*.
- Where an array is used as input or output, replace *Array* by *Array<T>*.
- In the source file, every function must be declared as a template of type *T*.
- The functions are now not a member of *Array* anymore but a member of *Array<T>*.
- Further replace all references to *Point* with *T*.
- Finally where an array is used as input or output, replace *Array* by *Array<T>*.
- Don't forget that the test program should now include the source file instead of the header file. Therefore, the source file should now also include `#ifndef/#define/#endif` statements.

In the test program create an array of points and test it:

```
Array<Point> points(size);
```

Tip: by placing the following code at the end of the array header file, but before the header file's `#endif`, the client can keep including the header file for template classes instead of the source file. Can you explain how this works?:

```
#ifndef Array_cpp       // Must be the same name as in source file #define
#include "Array.cpp"
#endif
```

4.2b - Advanced Templates

Exercise 1: Static Variable for Array Default Size

Static variables, which are shared between all instances of a class, behave slightly different with templates. We are going to test this with a static variable in the *Array* class that indicates the default array size when using the default constructor.

- Add a static data member to the *Array* class indicating the default size.
- Initialise this static in the source file to a value.
- Also add static functions to set and get the default size.
- In the default constructor, use the static default size variable instead of a literal value to set the array size.

Test the following code in the main program:

```
Array<int> intArray1;
Array<int> intArray2;
Array<double> doubleArray;

cout<<intArray1.DefaultSize()<<endl;
cout<<intArray2.DefaultSize()<<endl;
cout<<doubleArray.DefaultSize()<<endl;

intArray1.DefaultSize(15);

cout<<intArray1.DefaultSize()<<endl;
cout<<intArray2.DefaultSize()<<endl;
cout<<doubleArray.DefaultSize()<<endl;
```

What values are printed? Can you explain the result?

Exercise 2: Numeric Array (generic inheritance)

With templates it is possible to call functionality on the template argument. But when you do this, you limit the types you can use as the template argument because they have to support that functionality. Thus when adding numeric functionality to the *Array* class like calculating the sum, you would limit the types possible as the template argument to types that support numeric operations.

It is better to leave the *Array* class as generic as possible so it can be used in various situations. Numeric functionality can then be put in a derived class which itself is also a template (generic inheritance) as shown in Figure 2.

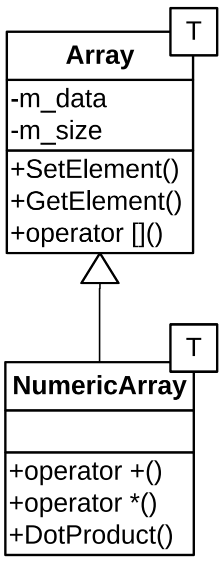

Figure 2: Numeric array using generic inheritance

In this exercise we are going to create a *NumericArray* derived from *Array*.

- Add a new source- and header file for a *NumericArray* class to your project.
- Create a template class called *NumericArray* and derive it from the *Array* class using generic inheritance.
- Since they are not inherited, create proper constructors, destructor and assignment operator and call the base class where appropriate.
- Add the following numeric functionality:
  - An `operator *` to scale the elements of the numeric array by a factor.
  - An `operator +` to add the elements of two numeric arrays. Throw an exception if the two arrays have not the same size.
  - A function to calculate the dot product. The dot product is defined as:

$$a.b = \sum_{i=1}^{n} a_i.b_i = a_1.b_1 + a_2.b_2 + \ldots + a_n.b_n$$

Change the main program to test the numeric array. What assumptions do you make about the type used as template argument? Can you create a numeric array with *Point* objects?

Exercise 3: Point Array (concrete inheritance)

You can create an array of points by directly instantiating the *Array* class for the *Point* class. This works well but we can't add points specific functionality to the *Array* class in this way.

A solution is to create a derived class of which the template argument is fixed to a specific type (concrete inheritance) as shown in Figure 3.

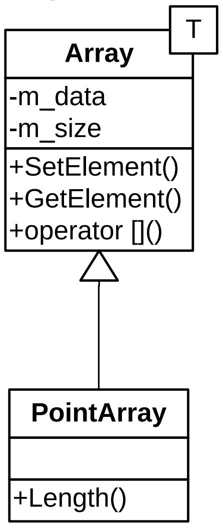

Figure 3: Point array using concrete inheritance

In this exercise we will create a *PointArray* which is derived from *Array* with the template argument set to *Point*.

- Add a new source- and header file for the *PointArray* class to your project.
- Create a regular class called *PointArray* which is derived from *Array*. The template argument given to *Array* is *Point*.
- Since they are not inherited, create proper constructors, destructor and assignment operator and call the base class where appropriate.
- Now we can add functionality specific for a point array. For example add a *Length()* function that returns the total length between the points in the array.
- Change the main program to test the point array.

Exercise 4: Stack Class (composition)

In this exercise we will make a *Stack* class. For the data storage we can use the *Array* class. Deriving from *Array* is not a good idea because there is no relation between Array and Stack. The stack would then inherit indexed operations which should not be a functionality of a stack. But we can use the *Array* class as a data member as shown in Figure 4:

- Add a new source and header file for the *Stack* class to your project.
- Create a template class called *Stack*. It is not a derived class but it uses an *Array* as data member. You also need a data member for the current index in the array.
- Create the regular constructors, destructor and assignment operator.
- Add a *Push()* function. It should store the element at the current position in the embedded array. Increment the current position afterwards. There is no need for checking the current index because the array will throw an exception when the stack is full. Make sure the current index is not changed when the *Array* class threw an exception.
- Add a *Pop()* function that decrements the current position and then returns the element at that position. Make sure the current index is not changed when the *Array* class throws an exception.
- Change the main function to test the stack.

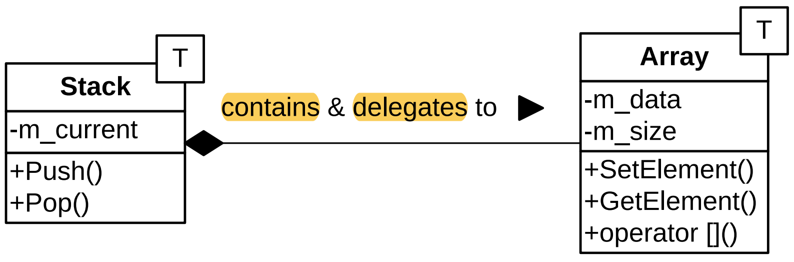

Figure 4: Stack class using composition

Using another class as data member is called composition. In this case the *Stack* class uses internally an *Array* class. Forwarding functionality to another class is called delegation. Here the *Stack* class delegates the storage of elements to the *Array* class.

Exercise 5: Layering Exceptions

When you use the *Stack* class and the stack is full or empty when pushing or popping an element you get an exception from the *Array* class. Since the user of the *Stack* class does not know that an *Array* class is used internally, you don't want that the client must know about the array exceptions. Thus the array exception must be translated to a stack exception.

- The stack exception classes can be implemented in a header file only for simplicity.
- Create a *StackException* base class and
  a *StackFullException* and *StackEmptyException* derived class.
- In the *Push()* function of stack, place the code in a *try* ... *catch* block and catch the array exception. In the catch handler, throw a *StackFullException* exception.
- In the *Pop()* function of stack, place the code in a *try* ... *catch* block and catch the array exception. In the catch handler, throw a *StackEmptyException* exception. Also set the current index back to 0.
- Change the test program so it catches the stack exception.

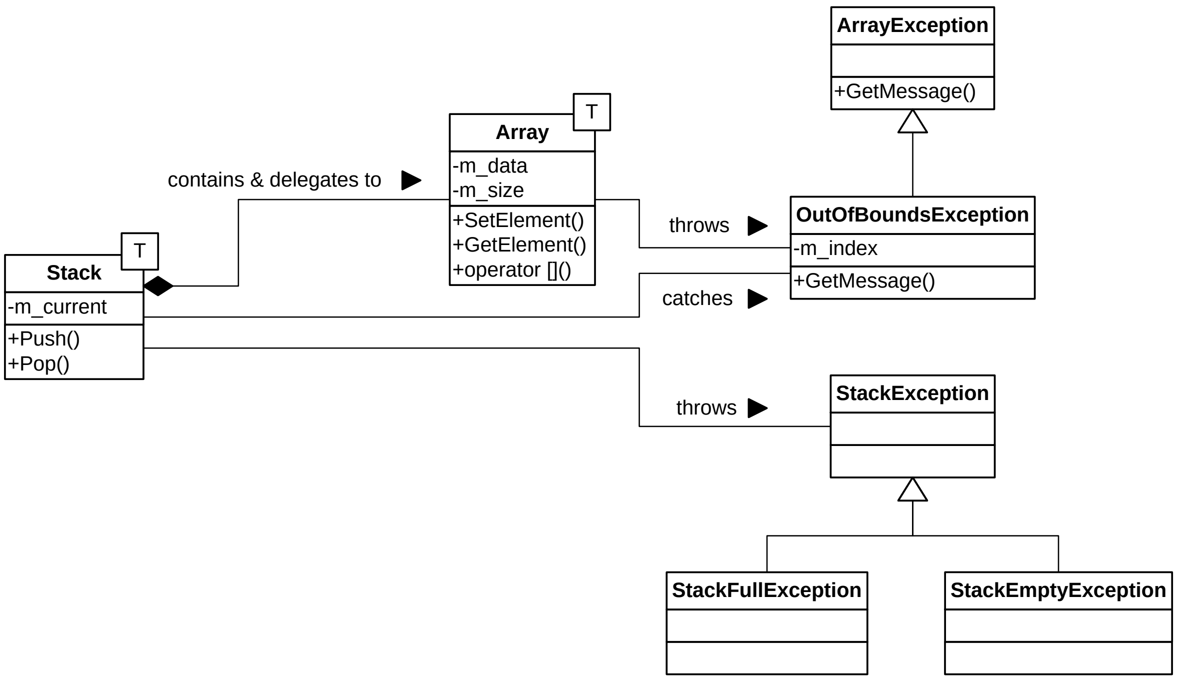

Figure 5: Layering Exceptions

Exercise 6: Value Template Arguments

Not only generic types can act as template variable. You can also use an *int* value as template variable. This value can then be used inside the class. Change the *Stack* class so that it uses a value template argument to set the stack size (remove the constructor with size):

```
template <typename T, int size> class Stack {};
```

Note that now only stacks with the **same** size template argument can be copied/assigned.