Part E – Polymorphism

Templates

Workshop 9 (1.0 in_lab)

In this workshop, you will work with a templated class that will allow for its member data and functions to operate on types supplied through a parameter list. The class in question will be a simplistic Calculator that works with arrays of numbers. The type of the numbers and the number of indices in the array will differ depending on what parameters are given to the Calculator class.

Learning Outcomes

Upon successful completion of this workshop, you will have demonstrated the abilities

- To code a templated class
- To specify the type to implement in a call to a templated class
- To specify a constant numeric parameter to a templated class
- To describe to your instructor what you have learned in completing this workshop

Submission Policy

The workshop is comprised of only 1 section;
in-lab - 100% of the total mark
To be completed before the end of the lab period and submitted from the lab.
The *in-lab* section is to be completed after the workshop is published, and before the end of the lab session. The *in-lab* is to be submitted during the workshop period form the lab.
If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. You must be present at the lab in order to get credit for the *in-lab* portion.
If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.
You are responsible to back up your work regularly.
Ask your professor if there are any additional requirements for your specific section.

Citation and Sources

When submitting the Lab part of this workshop, Project and assignment deliverables, a file called sources.txt must be present. This file will be submitted with your work automatically.
You are to write either of the following statements in the file "sources.txt":
I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.
Then add your name and your student number as signature
OR:
Write exactly which part of the code of the workshops or the assignment are given to you as help and who gave it to you or which source you received it from.
You need to mention the workshop name or assignment name and also the file name and the parts in which you received the code for help.
Finally add your name and student number as signature.
By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrong doing.

Late Submission Penalties:

-*In-lab* portion submitted late
0 for *in-lab*.

Workshop Due Dates

You can see the exact due dates of all assignments by adding -due after the submission command:
Run the following script from your account (use your professor's Seneca userid to replace profname.proflastname, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 244/NXX/WS09/in_lab -due<ENTER>
```

Compiling and testing your program

All your code should be compiled using this command on matrix:

```
g++ -Wall -std=c++11 -o ws (followed by your .cpp files)
```

After compiling and testing your code, run your program as follows to check for possible memory leaks: (assuming your executable name is “ws”)

```
valgrind ws <ENTER>
```

In-Lab (100%)

In this workshop we will be writing a simplistic and naïve **Calculator** class that can perform the cardinal mathematic operations of addition, subtraction, multiplication and division between two arrays of numbers. In addition to this Calculator will be a **templated class** so it may work with different numeric types.

Calculator Module

The **Calculator** module will be only module in this workshop and it will only be composed of the header file due to being a template. As such all definitions will be placed in **Calculator.h**.

Create a templated class called Calculator with the following template parameters:

Template Parameters:

Calculator will have two template parameters:

1. A **generic type** (T type for example) that will be representative type of our arrays of numbers
2. An **integer parameter** (int N for example) that will be used to determine the size of the array use in conjunction with our calculator.

Private members:

A **Calculator** will have the following data members:

- `results`
  This is an **array of the generic type** from the template parameters

(e.g. type T) with an **array size of the integer parameter** from the template parameters (e.g. N). This array will store the results of our calculations.

PUBLIC MEMBERS:

Constructors/Destructors

Calculator objects will only have one constructor.

1. Default constructor – This constructor should set a **Calculator** to a safe empty state.

Other Members

The other member functions will be facilitating the mathematical operations the **Calculator** can perform or print out the current results. The **type T** will be used for the purpose of the template parameter in the following functions.

Note: The Calculator is meant to be naïve and as such edge cases not mentioned by the specs won't be required to be implemented. The Calculator is also meant to only work with one generic type.

```
void add(const T*, const T*)
```

The add function will take two pointers of type T, these pointers will be pointing to the **first index of an array (in other words an array will be passed in to this param)**. Add the values of each index and the result of that will be stored in the **results** array in the respective index. **It can be assumed that the supplied arrays from the parameters will always be the same length as the results array.**

For example, given arr1 with values [1,2] and arr2 with values [2,1] by adding them together the Calculator's results array will then store [3,3].

```
void subtract(const T*, const T*)
```

The subtract function will take two pointers of type T, these pointers will be pointing to the **first index of an array**. Subtract the values of the first array at each index and with the values from the second array. The result of that will be stored in the **results** array in the respective index. **It can be assumed that the**

supplied arrays from the parameters will always be the same length as the results array.

For example, given arr1 with values [1,2] and arr2 with values [2,1] by subtracting them together the Calculator's results array will then store [-1,1].

```
void multiply(const T* arr1, const T*)
```

The multiply function will take two pointers of type T, these pointers will be pointing to the **first index of an array**. Multiply the values of the first array at each index and with the values from the second array. The result of that will be stored in the **results** array in the respective index. **It can be assumed that the supplied arrays from the parameters will always be the same length as the results array.**

For example, given arr1 with values [1,2] and arr2 with values [2,1] by multiplying them together the Calculator's results array will then store [2,2].

```
void divide(const T* arr1, const T*)
```

The divide function will take two pointers of type T, these pointers will be pointing to the **first index of an array**. Divide the values of the first array at each index and with the values from the second array. The result of that will be stored in the **results** array in the respective index. **It can be assumed that the supplied arrays from the parameters will always be the same length as the results array.**

For example, given arr1 with values [2.2,4.4] and arr2 with values [2.0,2.0] by multiplying them together the Calculator's results array will then store [1.1,2.2].

```
ostream& display(ostream& os) const;
```

This member function will display the current values of the **results** array. It will print them all in the same line separated by commas. Return **os** at the end of the function.

For example, if the **results** array had 4 numbers: [1,3,5,9] it would be displayed as:

```
1,3,5,9<newline>
```

Refer to the sample output for details.

Member Operator Overloads

There are four member operator overloads to implement for the Calculator. Consider using the previously established functions to provide the mechanism for these overloads.

```
Calculator& operator+=(const T*)
```

This function will take in a pointer of type T that points to the first index of an array. In contrast to the add function that added two parameters and stored that in the **results** array, this operator will use the **results** array as one of the operands. It will add the values in the parameter to the values in the **results** array and store the result of this operation in the **results** array.

It will lastly return the current object akin to the canonical += operator overloads.

```
Calculator& operator-=(const T*)
```

This function will take in a pointer of type T that points to the first index of an array. In contrast to the subtract function that subtracted two parameters and stored that in the **results** array, this operator will use the **results** array as one of the operands. It will subtract the values in the parameter from the values in the **results** array and store the result of this operation in the **results** array.

It will lastly return the current object akin to the canonical -= operator overloads.

```
Calculator& operator*=(const T*)
```

This function will take in a pointer of type T that points to the first index of an array. In contrast to the add function that multiplied two parameters and stored that in the results array, this operator will use the **results** array as one of the operands. It will multiply the values in the parameter with the values in the **results** array and store the result of this operation in the **results** array.

It will lastly return the current object akin to the canonical *= operator overloads.

```
Calculator& operator/=(const T*)
```

This function will take in a pointer of type T that points to the first index of an array. In contrast to the add function that divided two parameters and stored that in the results array, this operator will use the **results** array as one of the operands. It will divide the values in the parameter from the values in the **results** array and store the result of this operation in the **results** array.

It will lastly return the current object akin to the canonical /= operator overloads.

Note: Consider looking at the main to see how these overloads are used.

IN-LAB MAIN MODULE

```
/***********************************************************************
// OOP244 Workshop 9: Templates
// File CalculatorTester.cpp
// Version 1.0
// Date       2019/11/24
// Author     Hong Zhan (Michael) Huang
// Description
// Tests the Calculator template and its functions
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
// Michael
/////////////////////////////////////////////////////////////////
***********************************************************************/

#include <iostream>
#include <iomanip>
#include "Calculator.h"
using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
  for (int i = 0; i < len; i++, cout << ch);
  return cout;
}
ostream& number(int num) {
  cout << num;
  for (int i = 0; i < 9; i++) {
    cout << " - " << num;
  }
  return cout;
}

int main() {

  cout << "Create an int and double calculator" << endl;
  line(64, '-') << endl;
  number(1) << endl;
  Calculator<int, 5> ical;
  Calculator<double, 5> dcal;
```

```cpp
ical.display(cout);
dcal.display(cout) << endl;

cout << "Perform add function on each calculator" << endl;
line(64, '-') << endl;
number(2) << endl;
int arr1[5] = { 1, 2, 3, 4, 5 };
ical.add(arr1, arr1);
double arr2[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
dcal.add(arr2, arr2);
ical.display(cout);
dcal.display(cout) << endl;

cout << "Perform subtract function on each calculator" << endl;
line(64, '-') << endl;
number(3) << endl;
int arr3[5] = { 5, 4, 3, 2, 1 };
double arr4[5] = { 5.5, 4.4, 3.3, 2.2, 1.1 };
ical.subtract(arr3, arr1);
dcal.subtract(arr4, arr2);
ical.display(cout);
dcal.display(cout) << endl;

cout << "Perform multiply function on each calculator" << endl;
line(64, '-') << endl;
number(4) << endl;
ical.multiply(arr1, arr1);
dcal.multiply(arr2, arr2);
ical.display(cout);
dcal.display(cout) << endl;

cout << "Perform divide function on each calculator" << endl;
line(64, '-') << endl;
number(5) << endl;
ical.divide(arr1, arr1);
dcal.divide(arr2, arr2);
ical.display(cout);
dcal.display(cout) << endl;

cout << "Perform operators +=, -=, *=, /= on each calculator" << endl;
line(64, '-') << endl;
number(6) << endl;
cout << "ical's initial state: "; ical.display(cout);
cout << "ical += 1,2,3,4,5: ";
ical += arr1;
ical.display(cout);
cout << "ical -= 1,2,3,4,5: ";
ical -= arr1;
ical.display(cout);
cout << "ical *= 1,2,3,4,5: ";
ical *= arr1;
ical.display(cout);
cout << "ical /= 1,2,3,4,5: ";
ical /= arr1;
ical.display(cout) << endl;

cout << "dcal's initial state: "; dcal.display(cout);
cout << "dcal += 1.1,2.2,3.3,4.4,5.5: ";
cout << fixed << setprecision(2);
```

```
  dcal += arr2;
  dcal.display(cout);
  cout << "dcal -= 5.5,4.4,3.3,2.2,1.1: ";
  dcal -= arr4;
  dcal.display(cout);
  cout << "dcal *= 1.1,2.2,3.3,4.4,5.5: ";
  dcal *= arr2;
  dcal.display(cout);
  cout << "dcal /= 5.5,4.4,3.3,2.2,1.1: ";
  dcal /= arr4;
  dcal.display(cout);
  return 0;
}
```

EXECUTION EXAMPLE Red values are user entry

```
Create an int and double calculator
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
0,0,0,0,0
0,0,0,0,0

Perform add function on each calculator
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
2,4,6,8,10
2.2,4.4,6.6,8.8,11

Perform subtract function on each calculator
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
4,2,0,-2,-4
4.4,2.2,0,-2.2,-4.4

Perform multiply function on each calculator
----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
1,4,9,16,25
1.21,4.84,10.89,19.36,30.25

Perform divide function on each calculator
----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
1,1,1,1,1
1,1,1,1,1

Perform operators +=, -=, *=, /= on each calculator
----------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
ical's initial state: 1,1,1,1,1
ical += 1,2,3,4,5: 2,3,4,5,6
ical -= 1,2,3,4,5: 1,1,1,1,1
ical *= 1,2,3,4,5: 1,2,3,4,5
ical /= 1,2,3,4,5: 1,1,1,1,1
dcal's initial state: 1,1,1,1,1
dcal += 1.1,2.2,3.3,4.4,5.5: 2.10,3.20,4.30,5.40,6.50
dcal -= 5.5,4.4,3.3,2.2,1.1: -3.40,-1.20,1.00,3.20,5.40
dcal *= 1.1,2.2,3.3,4.4,5.5: -3.74,-2.64,3.30,14.08,29.70
dcal /= 5.5,4.4,3.3,2.2,1.1: -0.68,-0.60,1.00,6.40,27.00
```

IN-LAB SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload Calculator headerfile and the CalculatorTester.cpp program to your matrix account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace profname.proflastname, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 244/NXX/WS09/in_lab<ENTER>
```

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.