University of Calgary
Department of Electrical and Computer Engineering

Principles of Software Development – ENSF 480
Lab 3 – Fall 2023

Written by: M. Moussavi, PhD, PEng

In this lab, you are allowed to work with a partner, which means only groups of two are allowed (groups of three or more **are NOT allowed**). **If you decide to work with a partner, please submit only one lab report with both names. Submitting two lab reports with the same content will be considered as copies and is considered as plagiarism.**

Notes:

- **Some material related to exercise B and exercise C might be discussed during the week of Oct 11.**

Marking Scheme:

Exercise A	**10 marks**
Exercise B	**10 marks**
Exercise C	**20 marks**

Total marks: 40

Due Dates:

Lab section 1 (B01): Monday, Oct 9, before 2:00 PM
Lab section 2 (B02): Wed, Oct 11, before 2:00 PM
Lab section 3 (B03): Friday, Oct 13, before 2:00 PM

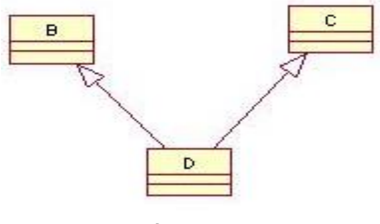
Introduction:

The objective lab is to understand the low-level design concepts of multiple inheritance, container classes and Iterator classes.

Exercise A - Multiple-Inheritance (10 marks)

This exercise is the continuation of exercise B in Lab 2. You have already completed exercise B in lab 2 and in this lab, you will add more code to your existing files in lab 2.

C++ allows a class to have more than one parent:



This is called multiple inheritance. For example, if we have two classes called B and C, and class D is derived from both classes (B and C), we say class D has two parents (multiple inheritance). This is a powerful feature of C++ language that other languages such as Java do not support. For the details, please refer to your class notes.

What to Do

In this exercise, you should add two new classes to your program completed in lab 2, exercise B:

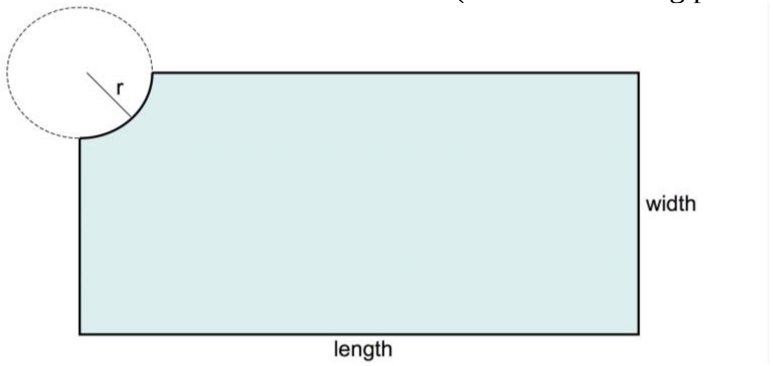
Class Circle:

This class is supposed to be derived from class Shape, and should have a data member radius. In addition to its constructor, it should support the similar functions as class `Rectangle`, such as `area`, `perimeter`, `get`, `set`, and `display`.

You should create two files for this class: called `circle.h and circle.cpp`

Class CurveCut:

CurveCut represents a shape that needs properties of class Rectangle and class Circle. In fact, it's a rectangle that its left top corner can have an arch-form cut (see the following picture).



This class must be derived from class Rectangle and class Circle. Where the origins (x and y coordinates) of both shapes are the same. This class, in addition to a constructor, should support the following functions:

- `area` – that calculates the highlighted area of the above figure.
- `perimeter` – that calculates and returns the perimeter of highlighted areas.
- `display` – that displays the name, x, y coordinates of the origin of the shape, width, and length (as shown in the picture above), and radius of the cut, in the following format:

```
CurveCut Name:
X-coordinate:
Y-coordinate:
Width:
Length:
Radius of the cut.
```

Note: The radius of the circle must be always less than or equal to the smaller width and length. Otherwise, the program should display an error message and terminate.

You should also add some code to function `run` in class `GraphicsWorld` to:

- test the member functions of class CurveCut.
- use an array of pointers to Shape objects, where each pointer points to a **different object** in the shapes hierarchy. Then test the functions of each class again.

A sample code segment that you can use to test your program is given in the following box (you can add more codes to this function if you want to show additional features of your program).

```
void GraphicsWorld::run(){
    /***************************ASSUME CODE SEGMENT FOR EXERCISE A IS HERE ************************/
#if 0
     cout << "\nTesting Functions in class Circle:" <<endl;
     Circle c (3, 5, 9, "CIRCLE C");
     c.display();
     cout << "the area of " << c.getName() <<" is: "<< c.area() << endl;
     cout << "the perimeter of " << c.getName() << " is: "<< c.perimeter() << endl;
     d = a.distance(c);
     cout << "\nThe distance between rectangle a and circle c is: " <<d;

     CurveCut rc (6, 5, 10, 12, 9, "CurveCut rc");
      rc.display();
      cout << "the area of " << rc.getName() <<" is: "<< rc.area();
      cout << "the perimeter of " << rc.getName() << " is: "<< rc.perimeter();
      d = rc.distance(c);
      cout << "\nThe distance between rc and c is: " <<d;


      // Using array of Shape pointers:
      Shape* sh[4];
      sh[0] = &s;
      sh[1] = &a;
      sh [2] = &c;
      sh [3] = &rc;
      sh [0]->display();
      cout << "\nthe area of "<< sh[0]->getName() << "is: "<< sh[0] ->area();
      cout << "\nthe perimeter of " << sh[0]->getName () << " is: "<< sh[0]->perimeter();
      sh [1]->display();
      cout << "\nthe area of "<< sh[1]->getName() << "is: "<< sh[1] ->area();
      cout << "\nthe perimeter of " << sh[0]->getName () << " is: "<< sh[1]->perimeter();
      sh [2]->display();
      cout << "\nthe area of "<< sh[2]->getName() << "is: "<< sh[2] ->area();
      cout << "\nthe circumference of " << sh[2]->getName ()<< " is: "<< sh[2]->perimeter();
      sh [3]->display();
      cout << "\nthe area of "<< sh[3]->getName() << "is: "<< sh[3] ->area();
      cout << "\nthe perimeter of " << sh[3]->getName () << " is: "<< sh[3]->perimeter();

      cout << "\nTesting copy constructor in class CurveCut:" <<endl;
      CurveCut cc = rc;
      cc.display();

      cout << "\nTesting assignment operator in class CurveCut:" <<endl;
      CurveCut cc2(2, 5, 100,  12, 9,  "CurveCut cc2");
      cc2.display();
      cc2 = cc;
      cc2.display();
#endif
}         // END OF FUNCTION run
```

What to Submit for Exercise A?

1. As part of your lab report (PDF file) submit: Copy of your complete source codes (.cpp and .h files), and the program output showing your code of this exercise works.
2. A zipped file with source file: All `.cpp` and `.h` files

Exercise B: Templates in C++ (10 marks)

Read This First:

What is a Template Function?

A template function is a function that one or more of its arguments types, and/or its return type are defined in a generic format that can be substituted with almost any actual built-in or user-defined type. These substitutions and creation of an instance of function is called instantiation of a template function. In the following example, function swap has been declared as a template function:

```
// function prototype
template  <class T > void swap (T* a, T*  b);

//function definition
template <class T>  void swap (T* a, T* b){
  T temp ;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

The keywords in this code are bold. The remaining parts of the code are similar to regular C++ functions. As mentioned earlier, this generic or template definition, can be instantiated with different types. For example, the following code shows how the template function swap has been instantiated, and how its first call receives two integer pointers; while its second instance receives two float pointers. For more details about template function please refer to your lecture notes (slides) or a textbook such as C++ Primer, the latest version.

```
void main() {
   int n = 8, m = 6;
   float x = 4.5, y =5.5;
   swap (&n,&m);  //  instantiation for swapping two integer
   swap (&x, &y); //  instantiation for swapping two float
}
```

What is a Template Class?

Templates classes are usually used to design and develop container data structures, such as queues, stacks, linked lists, vectors and etc. The template and non-template classes behave essentially the same. Once the template class has been made known to the program, it can be used as a type specifier, the same as a non-template class. However, the only difference is that the use of a template class name must always include its parameter list enclosed by angle brackets (except within its own class definition). The following examples show the definition of a template class called Vector, and one of its member functions, getValue():

```
template <class T>
class Vector {
  public:
    Vector(int s);
    ~Vector();
    T getValue(int elem);
  private:
    T *array;
     int size;
};

template <class T>
T Vector<T>::getValue (int elem) {
```

```
return array[elem];
}
```

Sometimes a template function cannot provide a correct instance of the functions. In these cases, you should write a specialization of the function. For more details about the specialization of a template function, please refer to your lecture notes.

If there is a need to a specialization of a member function, it must be defined in the same file, after the definition of the template function.

<u>What is Template Class Instantiation?</u>

A template class can be instantiated by appending the full list of actual parameters enclosed by angle brackets to the template class name. The following example shows how the template class Vector can be instantiated.

```
 int main() {
    Vector <char> x (3);
    Vector <double> y (40);
     …
     return 0;
}
```

<u>What is an Iterator</u>?

In general, an iterator is an object that provides a general method of successively accessing each element of a container type such as vectors or lists. In other words, it's an object that enables a programmer to traverse linear containers such as arrays, vectors and lists.

For example, we can write an iterator class called ***VectIter***, that provides overloaded operators to allow access to the elements of the Vector. Assume we have an object of ***VectIter*** called ***iter*** that initially is associated with the first element of an object of class ***Vector*** called ***myVector***, then a statement such as:

```
cout  << iter++;
```

displays the value of the first element of the vector and then advances the iterator object to the next object in the second element of the vector. This means that you must overload operator ++ (postfix) in the class IntVectorIter for this purpose.

Or, a statement such as:

```
cout << ++iter;
```

should advance the iterator object to the next element of the vector and then display the value of the previous element. This means that you should overload operator ++ (prefix) in the class IntVectorIter for this purpose.

Similarly, the following statement:

```
cout  << iter--;
```

displays the value of the current element of the vector and then moves the iterator object to the previous object in the vector. It means that you should overload operator -- (postfix) in the class IntVectorIter for this purpose.

What to Do for Exercise B:

Download files `mystring2.h`, `mystring2.cpp`, and `iterator.cpp` from D2L. Read the `iterator.cpp` carefully to understand the detail of the code in this file.

Now, you should:

1. Write the definition of operator functions declared in class `Vector`, and class `VectIter` that is embedded in the class `Vector`.

2. Convert the `Vector` class to a Template class. A class that virtually can create vectors of different types.

3. The given main function is partially compiled by the conditional compilation directives. Means a major segment of the code is commented out by using conditional compilation directive `#if 0`. Once you are done with the conversion of the class vector to a template class, change the `#if 0` directive to `#if 1` to include this part for compilation, and test your template class Vector and its iterator operators for data type: `int, Mystring`, and `char*`.

4. Make sure your function `ascending_sort` in the file `iterator.cpp` works properly for all data types.

5. **Note:** you may need to make other changes to the given source code to get your template class to work perfectly.

What to Submit:

Submit your source files iterator.cpp, mystring2.h and mystring2.cpp as part of your lab report (PDF file), the program output that shows that your program works, and a zipped file that contains ALL your source files for exercise B.

Exercise C (20 marks)

The class `LookupTable` in this exercise is very similar to class `DictionaryList` in one of the previous labs with a few exceptions that will be explained in the following section.

What to Do

1. Download files `customer.h, customer.cpp, lookupTable.h`, mainLab3ExC.cpp from D2L, and use your source codes `mystring2.h` and `mystring2.cpp`, from exercise B.

2. Compile, run, and observe the program output. Up to this point, the given code only works for objects of class Customer and keys of integer type. But our intention is to convert the entire code to generic type, in a way that being able to have lookup table that can have different type of keys (usually integer, long, or C-string type, or Mystring type) and virtually data of any type.

3. Class `LookupTable` is similar to DictionaryList, with some exceptions:
    - First, the data member `keyM` and `DatumM` are now placed in a `struct` called `Pair`. Please notice that there is a constructor defined for `Pair`.
    - The second difference is that an `iterator` class is embedded in the `LookupTable`. The job of this class is to manage a set of overloaded operators.

4. Modify the code and convert the `LookupTable` class to a template class.

5. Modify the global functions `try_to_find()` and `print()` to template functions.

6. Compile and test your program for creating objects of `LookupTable.`

7. Uncomment the code related to testing objects of `LookupTables of <int, Mystring>`.

8. Compile and test your program to create objects of `LookupTable<int, Mystring>`

9. Uncomment the code related to testing `LookupTable <int, int>`

10. Compile and test your program to create objects of `LookupTable<int, int>`

What to Submit:

- *As part of your lab report (pdf file):*
    - *Copy of your source code: `LookupTable.h`, `mainLab3ExC.cpp`.*
    - *The programs output that shows your program works.*
- *A zipped file that contains ALL your source files for exercise C (.cpp and .h)*