3.4 - Simple Inheritance

Exercise 1: Colon Syntax

The colon syntax can improve the performance of constructors. To test this, make sure that you print some text in the point's constructors, destructor and also the assignment operator.

Now, execute the following code in the test program and count the number of point constructor, destructor and assignment calls:

```
Line l;
```

Now, change the constructors in the *Line* class to use the colon syntax to set the start- and end-point data members and run the test program again. Is the number of point constructor, destructor and assignment calls less than before?

Apply the colon syntax also for the *Point* class constructors and if applicable also for the *Circle* class.

Exercise 2: Creating Shape Base Class

It can be useful to create a hierarchy of related classes using base- and derived classes.

- Classes are related (same family)
- Common data and functionality can be put in a base class.
- You can work with derived classes as if it is the base class.

In this exercise we are going to transform the *Point* and *Line* class into a *Shape* hierarchy as shown in Figure 1.

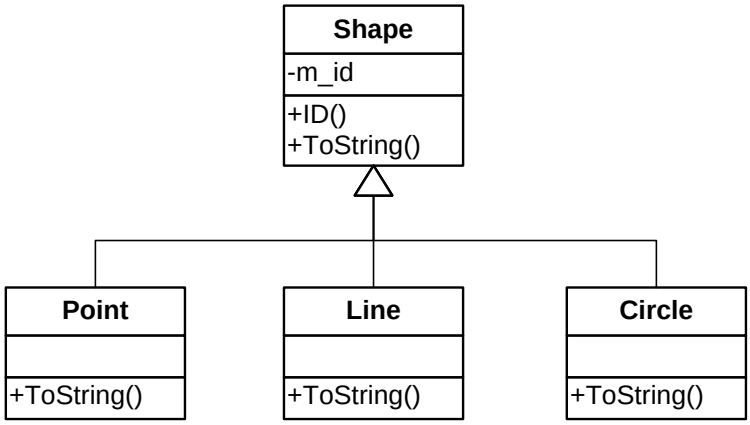

Figure 1: Shape Hierarchy

First create a *Shape* base class.

- Add a source- and header file for a *Shape* class.
- Add a data member for an *id* number of type *int*.
- Add a default constructor that initializes the *id* using a random number. You can use the *rand()* function from the "stdlib.h" header file.
- Add a copy constructor that copies the *id* member.
- Add an assignment operator that copies the *id* member.
- Add a *ToString()* function that returns the *id* as string e.g. "ID: 123".
- Add an *ID()* function the retrieve the *id* of the shape.

Next the *Point* and *Line* classes (and the *Circle* class if applicable) must derive from *Shape*.

- Add the *Shape* class in the inheritance list of the *Point*, *Line* and optionally the *Circle* class.
- The constructors of the *Point*, *Line* and optionally the *Circle* class should call the appropriate constructor in the *Shape*base class.
- The assignment operator should call the assignment operator of the *Shape* base class. *Otherwise the shape data will not be copied.*
- Finally add code to the main program to test inheritance:

```
Shape s;                            // Create shape.
Point p(10, 20);                    // Create point.
Line l(Point(1,2), Point(3, 4));    // Create line.

cout<<s.ToString()<<endl;           // Print shape.
cout<<p.ToString()<<endl;           // Print point.
cout<<l.ToString()<<endl;           // Print line

cout<<"Shape ID: "<<s.ID()<<endl;   // ID of the shape.
cout<<"Point ID: "<<p.ID()<<endl;   // ID of the point. Does this work?
cout<<"Line ID: "<<l.ID()<<endl;    // ID of the line. Does this work?

Shape* sp;                          // Create pointer to a shape variable.
sp=&p;                              // Point in a shape variable. Possible?
cout<<sp->ToString()<<endl;         // What is printed?

// Create and copy Point p to new point.
Point p2;
p2=p;
cout<<p2<<", "<<p2.ID()<<endl;      // Is the ID copied if you do not call
                                    // the base class assignment in point?
```

Answer the questions in the comments of the code above.

3.5 - Polymorphism

Exercise 1: Polymorphic *ToString()* Function

The *ToString()* functions of *Point* and *Line* override the *ToString()* from the *Shape* base class. We saw that we could put a*Point* in a *Shape** variable. But when calling the *ToString()* method on the *Shape** variable, the function in *Shape* was called instead the one in *Point*. To make the compiler generate the required code to find out what type of object the *Shape**variable is actually pointing to so it can call the right version; we need to declare the function as virtual.

Thus declare the *ToString()* function in the S*hape* class as virtual and test the program again. Is the *ToString()* function of *Point* called when you use a *Shape** that contains a *Point* now?

Exercise 2: Calling Base Class Functionality

The *ToString()* function of the *Shape* class is overridden in the derived classes. But for the derived class it is still possible to use the base class functionality. In the *ToString()* function of *Point* and *Line* we also want to incorporate the ID from the*Shape* base class.

- In the *ToString()* method of *Point*, call the *ToString()* method of the *Shape* base class: `std::string s=Shape::ToString();`
- Append the shape description string to the point description string before returning.
- Do this also for the *ToString()* function in the *Line* class (and *Circle* class).
- Test the application again. Is now the ID printed when printing a point or line?

Exercise 3: Virtual Destructors

When objects are removed from memory, the destructor is called. When a derived class destructor is called, it will automatically call the base class destructor. But when you have pointers to a base class, deleting objects might not be done correctly.

If not done already, print some text in the destructors of the *Shape*, *Point* and *Line* classes. Then test the following code:

```
Shape* shapes[3];
shapes[0]=new Shape;
shapes[1]=new Point;
shapes[2]=new Line;

for (int i=0; i!=3; i++) delete shapes[i];
```

Will the proper destructors (including the destructor of the *Shape* base class) be called?

In this case, the derived class destructor will only be called when the destructor is declared *virtual* in the base class. Do this in the *Shape* class and run the code again. Are the proper destructors called now?

Exercise 4: Abstract Functions

Sometimes functions in the base class are only there to be overridden in the derived class. Assume that you want to draw all the shapes using the following code:

```
Shape* shapes[10];

shapes[0]=new Line;
shapes[1]=new Point;
...
shapes[9]=new Line(Point(1.0, 2.5), Point(3.4, 5.2));

for (int i=0; i!=10; i++) shapes[i]->Draw();
for (int i=0; i!=10; i++) delete shapes[i];
```

Create the *Draw()* function in the *Shape* base class and override it in the derived classes (point, line and if present the circle class). Simulate drawing by just printing some text.

What implementation did you give the *Draw()* function in *Shape*? Shape is just an abstraction to work with various kinds of shapes like lines and circles. Shapes don't have a physical appearance. Therefore its *Draw()* function will have an empty implementation. But better is to give it no implementation at all by making it a pure virtual member function:

```
virtual void Draw()=0;
```

Do this in your code. Try to create an instance of the *Shape* class. Is this possible?
Now the *Shape* class is really an abstraction. You don't make shape instances but you can still create shape pointers that point to concrete shapes like point and line. The *Shape* class is now an abstract base class.

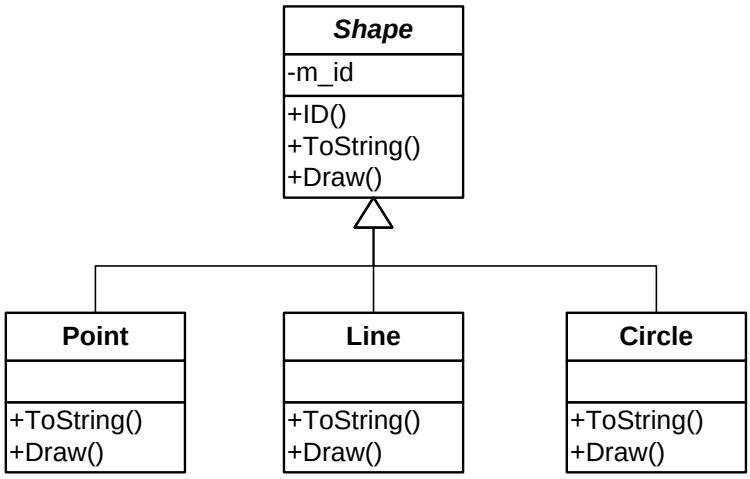

Figure 2: Abstract Shape base class with Draw() pure virtual member function

Exercise 5: Template Method Pattern

In this exercise we are going to create a *Print()* function that is printing the shape information to the *cout* object. The*Print()* function can use the *ToString()* to obtain the string to print.

You see that the implementation of *Print()* is in each derived class largely the same; calling *ToString()* and sending the result to *cout*. Since the *ToString()* function is polymorphic, we can use the polymorphic behavior in the *Print()* function of *Shape*.

Thus:

- Add a *Print()* function to the *Shape* class.
- In this function, call the *ToString()* function and send the result to the *cout* object.
- In the test program, create a point and line object and call the *Print()* function. Does it print the right information even when point and line do not have the *Print()* function?

You have now created a function for the base class (`Print()`) that does all the functionality common to all derived classes. Only the part of that function that is different for each derived class is delegated to a polymorphic function (`ToString()`). This mechanism is an often used design pattern called the “Template Method Pattern”.

3.6 - Exception Handling

Exercise 1: Bounds Checking Array

In the array class we created previously, the bounds checking was very basic. There was no error generated, but setting an element was ignored or the first element was returned. Obviously you want to know if there was an out of bounds error. This is possible by exception handling.

Change the *Array* class to throw exceptions:

- In the *GetElement()*, *SetElement()* and *index operator* throw *-1* if the index was too small or too big.
- In the main program, create an *Array* object and access an element that does not exist. Run the program. What happens?
- The exception must be caught, so place the code inside a *try ... catch* block that catches an *int*.
- In the catch handler, print an error message.

Exercise 2: Exception Objects

Throwing an *int* is an easy solution. But exception handling is also object oriented and allows us to throw an object.

In this exercise we will create an exception hierarchy with an *ArrayException* base class and an *OutOfBoundsException*derived class as shown in Figure 9:

- You can implement both exception classes in the header file for simplicity.
- Give the *ArrayException* an abstract *GetMessage()* function that returns a *std::string*.
- Give the *OutOfBoundsException* class a constructor with an *int* as argument that indicates the erroneous array index and store it in a data member.
- Override the *GetMessage()* function and let the implementation return a message string saying the given index is out of bounds.
- In the *Array* class, throw now a *OutOfBoundsException* object instead of an integer.
- Change the main program so that it catches the *ArrayException* base class and uses the *GetMessage()* function to display an error message.

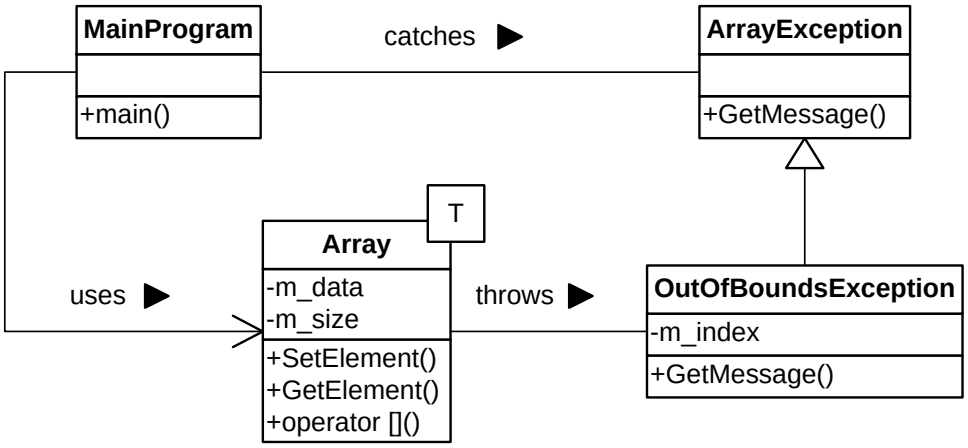

Figure 3: Array Exception