LABORATORY 8

Getting classy

OBJECTIVE

Objects are the basic unit of programming in object-oriented languages like `C++`. The `C++` construct for defining new types of objects is the class. A class is the “blueprint” from which an object can be created. This laboratory examines the basic mechanisms of class types by inspecting several different classes.

KEY CONCEPTS

- `class` construct
- Information hiding
- Encapsulation
- Data members
- Member functions
- Access specification
- Constructors
- Inspectors
- Mutators
- Facilitators

GETTING STARTED

- Using the procedures in the introductory laboratory handout, create the working directory `\cpplab` on the appropriate disk drive and obtain a copy of self-extracting archive `lab08.exe`. The copy should be placed in the `cpplab` directory. Execute the copy to extract the files necessary for this laboratory.
- Many of the activities that are performed in the laboratory can be done in groups but you should work the exercises yourself.

8.1 CLASSES

Objects are the fundamental units of programming in object-oriented languages such as C++. Objects are models of information and are implemented as software packages that contain or *encapsulate* both *attributes* and *behavior.* A *data abstraction* is a representation of information and the operations to be performed on it.

A fundamental type is a type that the programming language provides. For example, in C++, the type `int` and the operations on it are part of the language definition. That is, all C++ compilers must provide `int` objects. In addition to the fundamental types, C++ provides several mechanisms to define other types. These other types are called *programmer-* or *user-defined types*.

The `class` construct is the most important mechanism for defining new types. With this construct, software engineers can define encapsulated objects.

When defining a class, we typically use the information-hiding principle:

> All interaction with an object should be constrained to use a well-defined interface that allows underlying object implementation details to be safely ignored.

By following the information-hiding principle, we can create classes that tend to be reliable and easily reused.

To consider the basics of creating a new object type using the `class` construct, we first examine the declaration of a programmer-defined type we have used previously—`RectangleShape`. Whenever one is designing something, it helps to know how the thing will be used. Our goal in designing the class `RectangleShape` is to create a graphical object that is both easy to construct and simple to use.

The first thing to do when creating a class-type object is to determine the attributes of the object. For rectangle objects in a graphical display system, the necessary attributes are the window in which it will be displayed, its location within the display window, its color, and of course, its size. In C++, the attributes of a class-type object are referred to as the *data members*. The following are the declarations of the data members of `RectangleShape`:

```
SimpleWindow &Window;
float XCenter;
float YCenter;
color Color;
float Width;
float Height;
```

The second step when creating a class-type object is to determine the messages the object can receive and the operations that can be performed on the object. This portion of an object is called its behavior component. The behavior component of a class-type object is a collection of *member functions* that provide the ability to send messages to the object requesting it to perform some action. These member functions provide the public interface to achieve the desired behavior; other member functions are often present to assist the member functions that are part of the public interface.

For a rectangle shape in a windowed graphical display system, we can divide the messages that a rectangle needs to handle can be divided into three sets. One set of messages directs a `RectangleShape` object to return an attribute's value. The second set of messages directs a `RectangleShape` object to change the value of an attribute. The third set of messages directs a `RectangleShape` object to perform a service.

Messages that return the value of an attribute are called *inspectors*. For maximum flexibility, our definition of `RectangleShape` includes inspectors for all of the data members.

```
color GetColor() const;
void GetSize(float &Width, float &Height) const;
float GetWidth() const;
float GetHeight() const;
void GetPosition(float &XCoord, float &YCoord)
 const;
SimpleWindow& GetWindow() const;
```

Inspectors will have the qualifier `const` after their parameter list. The qualifier indicates that the member function does not modify the object. An inspector name usually begins with `Get`. Notice that the inspector `GetWindow()` returns a reference (indicated by the ampersand (`&`) after the class name). The ampersand tells the compiler that it should return a reference to the window that contains the rectangle. If the ampersand were omitted, the compiler would attempt to return a copy of the window that contains the rectangle. We really do not want to create another window, so returning a reference is the correct action.

Messages that change or set an attribute are called *mutators*. Again, to make the `RectangleShape` as flexible as possible, we will have mutators for the attributes that control the appearance and location of a `RectangleShape`.

```
void SetColor(const color &Color);
void SetPosition(float XCoord, float YCoord);
void SetSize(float Width, float Height);
```

There will be no mutator for setting the window display because a `RectangleShape` object when defined is associated with a particular display that cannot be changed.

There is one class member function that is neither an inspector nor a mutator. The draw message tells the rectangle to display itself in the window.

```
void Draw();
```

`Draw()` is an example of a facilitator. A *facilitator* performs a service.

In addition to the class member functions that process messages, a class definition will also specify class constructors. A *constructor* is a member function that initializes an object of that class. A constructor has the same name as the class. The appropriate constructor is invoked automatically when a `RectangleShape` object is defined.

The following `RectangleShape` constructor expects an initial value for each of the data members. Unlike other functions, a constructor does not have a return type.

```
RectangleShape(SimpleWindow &Window, float XCoord,
 float YCoord, const color &Color, float Width,
 float Height);
```

Besides listing the various members, the definition of a class also indicates which parts of a program can use the members. There are three kinds of access permissions: `public`, `private`, and `protected`. The general rule is

A `public` member has unrestricted access.

A `private` member can be accessed only by other members of the same class.

A `protected` member does not have unrestricted access. It can be accessed only by other members of the same class or classes derived from this class.

- Open the file `rect.h` to see the class definition for `RectangleShape`. It should resemble the following:

```
#ifndef RECTSHAPE_H
#define RECTSHAPE_H

#include "ezwin.h"
class RectangleShape {
   public:
      RectangleShape(SimpleWindow &Window,
       float XCoord, float YCoord,
       const color &Color,
       float Width, float Height);
      void Draw();
      color GetColor() const;
      void GetSize(float &Width,
       float &Height) const;
      void GetPosition(float &XCoord,
      float &YCoord) const;
      SimpleWindow& GetWindow() const;
      void SetColor(const color &Color);
      void SetPosition(float XCoord,
       float YCoord);
      void SetSize(float Width, float Height);
   private:
      SimpleWindow &Window;
      float XCenter;
      float YCenter;
      color Color;
      float Width;
      float Height;
};

#endif
```

- Examine the code.

- Explain to your laboratory instructor the purpose of the preprocessor directives `ifndef`, `define`, and `endif`. If you are unsure about any aspects of the definition of class `RectangleShape`, consult your laboratory instructor before proceeding. ✓

When defining a class, we give the `public` section first because the members in this section can be used by anyone. We give the `protected` section (if it exists) next. The `private` section comes last.

- Open the file `element.h`. The code should resemble the following:

```
#ifndef ELEMENT_H
#define ELEMENT_H

class Element {
   public:
      Element(int x, int y, int z);
      int GetX() const;
      int GetY() const;
      int GetZ() const;
      void SetX(int x);
      void SetY(int y);
      void SetZ(int z);
   private:
      int X;
      int Y;
      int Z;
};

#endif
```

- This file describes the interface for the class `Element` that represents an element of three-dimensional space. Identify the data members, inspectors, mutators, facilitators, and constructors of class `Element`. ✓
- Open the project `blue.ide`. Then open the file `blue.cpp` from that project. Examine the function `ApiMain()` and its helper functions `DisplayRectangleAttributes()` and `Convert()`. Observe that to access a member, we use the selection operator, which is the period (`.`).
- Build the project and examine the results.
- Modify the program by defining and drawing a second `RectangleShape` object `S`. Have `S` be different in size, color, and position from the `RectangleShape` object `R` already defined in the program. Make a second call to `DisplayRectangleAttributes()` to also display the characteristics of `S`. Demonstrate your modified program to the laboratory instructor. ✓

We will now test whether the access permissions associated with the `RectangleShape` are truly in effect. Rather than using the inspectors inside `DisplayRectangleAttributes()` to obtain the data member values, we will attempt to directly access the data members. For example, to get a copy of the color of the formal parameter `r` of `DisplayRectangleAttributes()`, assign local object `c` with the value `r.Color`.

- Compile your modified file. Examine the compiler messages. ✓
- Restore `blue.cpp` so that it uses the inspectors.

- Open the file `rect.cpp`. Examine the functions defined in that file. Observe that the class name and the scope resolution operator precede the member function name. Note that the scope resolution operator is a double colon (`::`). The member functions must be identified this way so that the compiler can tell that they belong to a class.

The first function defined in the file is a `RectangleShape` constructor. A constructor for a class is invoked when a object of that class is defined.

- To verify that the `RectangleShape` constructor is invoked when a `RectangleShape` object is defined, add an insertion statement in `RectangleShape`'s constructor that inserts the following message to the stream `cout`:

  ```
  RectangleShape constructor called.
  ```

  Also, add two insertion statements in the function `ApiMain()`. Add one before the definition of object `R` and one after it. Execute the modified program. Describe what is happening to the laboratory instructor. ✓
- Remove the insertion statements you added to `RectangleShape`'s constructor and to function `ApiMain()`.
- Observe that the constructor uses the available mutators for setting the attributes. The constructor uses an initialization list for specifying the data member `Window`.
- Modify the constructor so that mutators are not used. That is, set the values of the data members using assignment operations. Compile the file `rect.cpp`. Observe that a member function of a class, even a public member function, can access the private members. ✓

You may wonder why the constructor uses the mutators to set the values of the data members when the constructor could access the data members directly. The key idea is that all access to the data members should be through the same interface (i.e., member function). Thus a client user of a class changes the data members the same way a public member function of the class changes them. Using this strategy means that we can change the representation of a data member without having to modify every class member function that changes the value of the data member. Since all access to the data member is through the mutator, we need to change only the implementation of the mutator. This concept is probably fuzzy because you have not yet seen enough code to appreciate this very important design strategy. However, as you develop additional classes and modify existing classes, you will see that always using a mutator to change a data member value (whether the entity doing the changing is a client user of the class or a member function of the class) makes software more flexible, easier to maintain, and easier to modify.

- Restore `rect.cpp` so that the mutators are used.
- Add a `RectangleShape` member function `Double()` to the file `rect.cpp`. This function first doubles the width and height of the object and then draws the object. Have your function `Double()` use the `RectangleShape` inspector `GetSize()` to get copies of the current width and

height and then use mutator `SetSize()` to assign the new width and height values. Is `Double()` an inspector, mutator, or facilitator? Should the qualifier `const` be used?

- Modify `ApiMain()` so that `RectangleShape` object `R` has its size doubled. Also display its attributes after it has been doubled.
- Make the project `blue.ide`. Can you remove all errors regarding function `Double()`? Why?
- Modify the class definition of `RectangleShape` in `rect.h` to include a member function `Double()`. Remake and run the project `blue.ide`. ✓
- Close the `blue.ide` project.

8.2

IT'S ELEMENTAL

- Now open the project `element.ide` and its files `eletest.cpp` and `element.cpp`. Complete the member function bodies of the various member functions in `element.cpp`. In particular, use mutators for the `Element` constructor.
- Test your implementation by building and running the project. ✓
- Change the line in `eletest.cpp` so that instead of changing `OtherPoint`'s x-axis value, a change is made to `Center`'s x-axis value.
- Recompile `eletest.cpp`. How does the compiler know that it is wrong for `Center` to invoke its member function `SetX()`, but not wrong for it to invoke its member function `GetX()`?

A common graphical object is a line segment (see Figure 8.1). A line segment in three-dimensional space is specified by its three-dimensional endpoints.

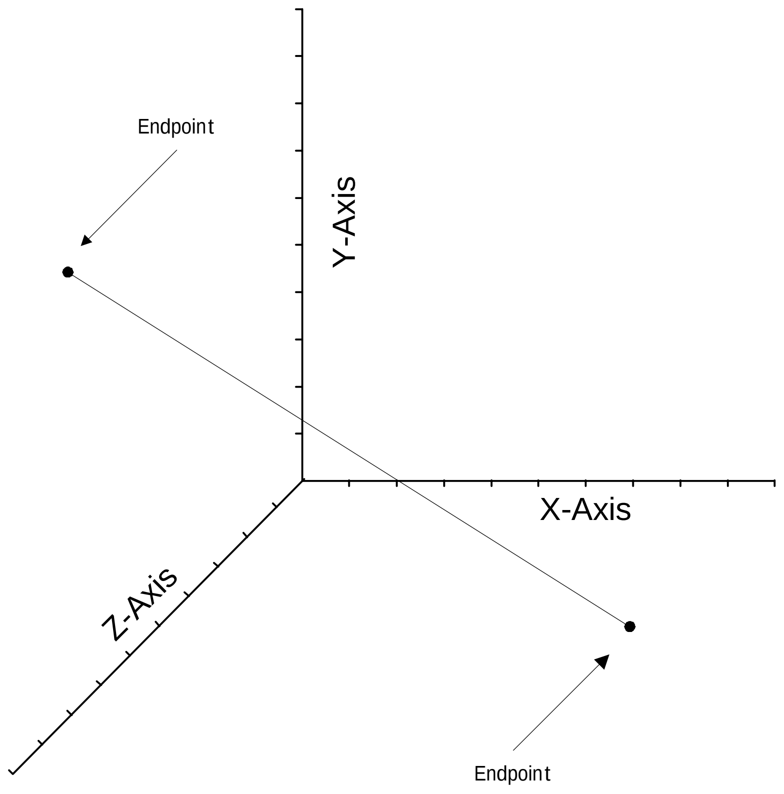
- Suppose you are to develop a class to represent three-dimensional line segments. What inspectors and mutators should be present? What facilitators might be helpful? Does it make a difference whether the class is supposed to represent mathematical or graphical objects? Should it make a difference? Discuss your answers with your laboratory instructor. ✓

8.3

FINISHING UP

- Copy any files you wish to keep to your own drive.
- Delete the directory `\cpplab`.
- Hand in your check-off sheet.

Figure 8.1
A line segment in 3-space


Endpoint
Y-Axis
X-Axis
Z-Axis
Endpoint