Learning Objectives: Polymorphism

- **Define polymorphism**
- **Explain how function overriding is an example of polymorphism**
- **Overload a function**
- **Override a function**
- **Use an abstract function as a form of polymorphism**

Function Overriding

What is Polymorphism?

Polymorphism is a concept in object-oriented programming in which a single interface takes different forms (polymorphism means "many forms"). Often this means similar operations are grouped together with the same name. However, these operations with the same name will produce different results. You have already encountered a few examples of polymorphism. Enter the following code into the IDE.

```
//add code below this line

int a = 5;
int b = 10;
cout << (a + b) << endl;

string c = "5";
string d = "10";
cout << (c + d) << endl;

bool e = true;
bool f = false;
cout << (e + f) << endl;

//add code above this line
```

Notice how the plus operator (+) can add together two numbers, concatenate two strings, and add two booleans. You have a single interface (the plus operator) taking different forms — one that works with integers, another that works with strings, and even one that works with booleans. This is an example of polymorphism.

▼ **Operator Overloading**

Because the plus operator can work with different forms, we can say that it is overloaded. C++ overloads this operator by default. However, a user cannot manually overload an operator.

challenge

Try these variations:

- Change your code to look like this:

```
//add code below this line

int a = 5;
string b = "true";
cout << (a + b) << endl;

//add code above this line
```

▼ **Why is there an error?**
Polymorphism allows C++ to use the plus operator with different data types, but that does not mean that the plus operator can be used with all data types. The example above causes an error message because the plus operator cannot be used with an integer and a string. There are limits to polymorphism.

- Change `string b = "true";` in the code above to `bool b = true;`.

▼ **Why does the code above work?**
Remember that a boolean value of `true` is equivalent to an integer value of `1`. This is why it is possible to add boolean and integer values together.

Function Overriding

Function overriding is another example of polymorphism that you have already seen. Overriding a function means that you have two functions with the same name, but they perform different tasks. Again you see a single interface (the function name) being used with different forms (the base class and the derived class). Create the following classes.

```
//add class definitions below this line

class Alpha {
  public:
    void Show() {
      cout << "I am from class Alpha" << endl;
    }
};

class Bravo: public Alpha {
  public:
    void Show() {
      cout << "I am from class Bravo" << endl;
    }
};

//add class definitions above this line
```

Then instantiate an `Alpha` object and call the `Show` function.

```
  //add code below this line

  Alpha test_object;
  test_object.Show();

  //add code above this line
```

As expected, the script prints `I am from class Alpha`. Now change the line of code in which you instantiate the object `test_object` to a Bravo object like below. Make no other changes and run the code again.

```
  Bravo test_object;
```

Now the script prints `I am from class Bravo`. The function call did not change, but the output did. A single interface (the `Show` function) works with multiple forms (the `Alpha` and `Bravo` data types). This is why function overriding is an example of polymorphism.

challenge

Try this variation:

- Create and overload the function `Hello` that prints `Hello from Alpha` and `Hello from Bravo` to the `Alpha` and `Bravo` classes

respectively. Then test the function on both class types by calling it on their respective objects.

▼ **Solution**

```
//add class definitions below this line

class Alpha {
  public:
    void Show() {
      cout << "I am from class Alpha" << endl;
    }

    void Hello() {
      cout << "Hello from Alpha" << endl;
    }
};

class Bravo: public Alpha {
  public:
    void Show() {
      cout << "I am from class Bravo" << endl;
    }

    void Hello() {
      cout << "Hello from Bravo" << endl;
    }
};

//add class definitions above this line

int main() {

  //add code below this line

  Alpha test_object; //Then test with Bravo test_object
  test_object.Hello();

  //add code above this line

  return 0;

}
```

Function Overloading

Function Overloading

Function overloading is another example of polymorphism. Function overloading occurs when you have a single function name that can take different sets of parameters. Imagine you want to write the function `Sum` that can sum up to three numbers. The math involved with three parameters is slightly different than two parameters, which is different from 1 parameter, etc. Traditionally, if you declare a function that takes three parameters but only pass two, C++ will throw an error message. Instead, let's create a class that has two `Sum` functions; one with two parameters and another with three parameters.

```
//add class definitions below this line

class TestClass {
  public:
    int Sum(int n1, int n2, int n3) {
      return n1 + n2 + n3;
    }

    int Sum(int n1, int n2) {
      return n1 + n2;
    }
};

//add class definitions above this line
```

Create an object of type `TestClass` and call both versions of the `Sum` function. Be sure you are passing three arguments for one function and two arguments for the other.

```
  //add code below this line

  TestClass tc;
  cout << tc.Sum(1, 2, 3) << endl;
  cout << tc.Sum(1, 2) << endl;

  //add code above this line
```

C++ looks at the number and types of arguments and, as long there is a matching function definition, runs the code without an error. Defining the same function with different sets of arguments is called overloading. It is also an example of polymorphism.

challenge

Try this variation:

- Continue to overload the `Sum` function such that it can take up to five numbers as parameters (which means 4 functions total). Be sure to test all possible function calls in `main`.

▼ **Solution**

```
//add class definitions below this line

class TestClass {
  public:
    int Sum(int n1, int n2, int n3, int n4, int n5) {
      return n1 + n2 + n3 + n4 + n5;
    }

    int Sum(int n1, int n2, int n3, int n4) {
      return n1 + n2 + n3 + n4;
    }

    int Sum(int n1, int n2, int n3) {
      return n1 + n2 + n3;
    }

    int Sum(int n1, int n2) {
      return n1 + n2;
    }
};

//add class definitions above this line

int main() {

  //add code below this line

  TestClass tc;
  cout << tc.Sum(1, 2, 3, 4, 5) << endl;
  cout << tc.Sum(1, 2, 3, 4) << endl;
  cout << tc.Sum(1, 2, 3) << endl;
  cout << tc.Sum(1, 2) << endl;

  //add code above this line

  return 0;

}
```

Overloading the Constructor

C++ will also allow you to overload the constructor so that objects are instantiated in a variety of ways. The `Person` class has a default constructor (no arguments) and a constructor with three arguments.

```
//add class definitions below this line

class Person {
  public:
    Person() {}

    Person(string na, int nu, string s) {
      name = na;
      number = nu;
      street = s;
    }

    string Info() {
      return (name + " lives at " + to_string(number) + ' ' +
        street + '.');
    }

  private:
    string name;
    int number;
    string street;
};

//add class definitions above this line
```

When you create a `Person` object with no arguments, the `Info` function still works. However, the information that is printed may look jarring since C++ will use left-over "junk" memory data to fill in for variables that are not initialized. You can also instantiate an object with three arguments. Like function overloading, constructor overloading is a form of polymorphism.

```
  //add code below this line

  Person p1;
  Person p2("Calvin", 37, "Main Street");
  cout << p1.Info() << endl;
  cout << p2.Info() << endl;

  //add code above this line
```

▼ Random Values

Unlike other programming languages, C++ does not automatically initialize variables that are not initialized by the user. This is done to preserve memory. Thus, you might get “randomly” generated data for certain uninitialized variables. Run the code several more times and you’ll notice different values will be printed.

challenge

Try these variations:

- Comment out both of the constructors.

```
//add class definitions below this line

class Person {
  public:
    //Person() {}

    //Person(string na, int nu, string s) {
      //name = na;
      //number = nu;
      //street = s;
    //}

    string Info() {
      return (name + " lives at " + to_string(number) + " "
        + street + ".");
    }

  private:
    string name;
    int number;
    string street;
};

//add class definitions above this line
```

Instantiate a `Person` object and call the `Info` function.

```
  //add code below this line

  Person p1;
  //Person p2("Calvin", 37, "Main Street");
  cout << p1.Info() << endl;
  //cout << p2.Info() << endl;

  //add code above this line
```

▼ **Why does this work?**

When you do not declare a constructor, C++ will use the default constructor and give each of the attributes their default value.

- Uncomment only the constructor with three arguments.

```
//add class definitions below this line

class Person {
  public:
    //Person() {}

    Person(string na, int nu, string s) {
      name = na;
      number = nu;
      street = s;
    }

    string Info() {
      return (name + " lives at " + to_string(number) + " "
        + street + ".");
    }

  private:
    string name;
    int number;
    string street;
};

//add class definitions above this line
```

Do not make any changes to the object instantiation in `main`.

▼ **Why is there an error?**

C++ automatically uses the default constructor when there are no constructors defined. If, however, a constructor exists, the object that

gets instantiated must contain the same number of arguments as specified by that constructor's parameters. Otherwise, an error will be produced.

Abstract Functions

Abstract Classes

Another form of polymorphism in C++ involves **abstract** functions. These functions, however, require knowledge of abstract classes. So before we continue the discussion on polymorphism, we need to first talk about abstract classes.

▼ **Concrete Classes**

Any class that is not an abstract class is considered to be a concrete class. You do not need to use a keyword to indicate that a class is concrete.

A defining characteristic of an abstract class is that an abstract class has at least one abstract, or pure **virtual**, function. An abstract function is a function that is defined as being equal to `0` in the base class, but is expected to be redefined in the derived class.

Abstract (Pure Virtual) Functions

When will there be a need for abstract functions? Abstract functions are used when the derived classes are expected to use a particular abstract function from the base class differently. Let's take a look at a classic example of the *Shape* class.

```
//add class definitions below this line

class Shape {
  public:
    virtual double Area() = 0;
};

//add class definitions above this line
```

As seen in the above code, the virtual function `Area` is defined as being equal to `0`. We do this because we expect classes that are derived from `Shape` to have `Area` functions that behave differently. For example, calculating the area of a `Triangle` object is different from calculating the area of a `Rectangle` object. Thus, we define `Area` as an abstract function in

the base class `Shape`. Note that when you redefine abstract functions in the derived classes, you **do not** include the `virtual` keyword nor assign the function to `0`. Let's define our classes further.

```
//add class definitions below this line

class Shape {
  public:
    virtual double Area() = 0;

    double GetBase() {
      return base;
    }

    void SetBase(double new_base) {
      base = new_base;
    }

    double GetHeight() {
      return height;
    }

    void SetHeight(double new_height) {
      height = new_height;
    }

  protected:
    double base;
    double height;
};

class Triangle : public Shape {
  public:
    Triangle(double b, double h) {
      base = b;
      height = h;
    }

    double Area() {
      return base * height / 2;
    }
};

class Rectangle : public Shape {
  public:
    Rectangle(double b, double h) {
      base = b;
```

```
      height = h;
    }

    double Area() {
      return base * height;
    }
};

//add class definitions above this line
```

You can see above that two classes are derived from `Shape`, the `Triangle` class and the `Rectangle` class. Also notice how `Shape` contains additional getters and setters as well as `protected` attributes. We encapsulate them as protected in order for our derived classes to access them. This way, we don't have to declare additional attributes in `Triangle` and `Rectangle`.

Next, let's test our code in `main`.

```
  //add code below this line

  Triangle t(4, 4);
  cout << t.Area() << endl;
  Rectangle r(4, 4);
  cout << r.Area() << endl;

  //add code above this line
```

As expected, the code returns the correct calculations for the `Triangle` and `Rectangle` objects.

challenge

Try this variation:

- Change `cout << t.Area() << endl;` to `cout << t.Shape::Area() << endl;`
- Replace the entire code in `main` with

```
//add code below this line

Shape s;

//add code above this line
```

▼ **Why are there errors?**

You cannot create an object of an abstract class nor can you call an abstract function. This is why using encapsulation and inheritance is important in redefining functions so that they can be used in derived classes. Because abstract functions get redefined and used differently, they are considered to be a concept of polymorphism.