Introduction to Classes

So far, the lessons have focused on presenting the basic syntax of C++. You've learned how to declare and use variables, control the flow of execution with loops and conditional processing, some basic ways to handle input and output and about pointers and arrays. This lesson introduces classes and objects. The use of objects in C++ defines the way programs are designed and written. Classes are a software construct that can be used to emulate a real world object. Classes encapsulate data and abilities. For instance, a software model of a car, a car "class", might contain data about the type of car and abilities such as accelerate or decelerate. A class is a programmer defined data type that has data, its members, and abilities, its methods. An object is a particular instance of a class. This is best understood by an analogy with a built in data type such as int.

```
int x;
```

Declares x to be a variable of type int.

```
Car impala;
```

Declares impala to be an object of class Car.

Defining Classes
A class is defined by using the keyword class followed by a programmer-specified name followed by the class definition in braces. The class definition contains the class members, its data, and the class methods, its functions. As an example, let's construct a "Dog" class that will be a model of the real world pets many of us have.

```
class Dog {
public:
   void setAge(int age);
   int getAge();
   void setWeight(int weight);
   int getWeight();
   void speak();
private:
   int age;
   int weight;
};
```

This simple example illustrates several important concepts. First, the keyword private indicates that the two members, age and weight, cannot be directly accessed from outside of the class. The keyword public indicates that the methods, setAge, getAge, setWeight, getWeight and speak, can be called from code outside of the class. That is, they may be called by other parts of a program using objects of this class. This technique of allowing access and manipulation of data members only through methods is referred to as data hiding. The interface to the class is public and the

data is private. Public interface, private data is a key concept when designing classes. Data hiding will be discussed more in a later section of this article. Also, note that four of the methods, setAge, getAge, setWeight and getWeight, are involve reading or updating members of the class. Methods used to set or get members are called accessor methods or accessors.

In the above class definition, the methods are declared but not defined. That is, an implementation for each method must be written.

```
class Dog {
public:
    void setAge(int age);
    int getAge();
    void setWeight(int weight);
    int getWeight();
    void speak();
private:
    int age;
    int weight;
};

void Dog::setAge(int age)
{
    this->age = age;
}

int Dog::getAge()
{
    return age;
}

void Dog::setWeight(int weight)
{
    this->weight = weight;
}

int Dog::getWeight()
{
    return weight;
}

void Dog::speak()
{
    cout << "BARK!!" << endl;
}
```

There are a few more important things to notice here. First, since the methods are implemented outside of the class definition, they must be identified as belonging to

that class. This is done with the scope resolution operator, "::". It identifies each method, for example, getAge, as belonging to the class Dog. Second, every object has a special pointer call "this", which refers to the object itself. So the members of the Dog class can be referred to as this->age or this->weight, as well as, age or weight. If there is no ambiguity, no qualification is required. So in the getWeight method, "weight" can be used instead of "this->weight". In the setWeight method an ambiguity exists. Since the parameter passed is "weight" and there is a class member "weight", the "this" pointer must be used. Finally, a note about syntax. If "this" is a pointer to a class, then the member selection operator, "->", can be used to access the contents of its members.

Constructors and Destructors

Each class also has a special method, the constructor, that is called when an object of the class is instantiated (created). The constructor can be used to initialize variables, dynamically allocate memory or setup any needed resources. Another special method, the destructor, is called when an object is destroyed. An object is destroyed when it goes out of scope. If an object is created within a function, it will go out of scope when the function exits. Since your program is the "main" function, all its objects go out of scope when the program ends. Scope is described fully in a later lesson. The destructor is used to free any memory that was allocated and possible release other resources. Here is the Dog class with a constructor and a destructor added. Since they are given on the last page, the implementations of all member functions are not reproduced here.

```
class Dog {
public:
   Dog();      //Constructor
   ~Dog();    //Destructor
   void setAge(int age);
   int getAge();
   void setWeight(int weight);
   int getWeight();
   void speak();
private:
   int age;
   int weight;
};

Dog::Dog()
{
   age = 0;
   weight = 0;
   cout << "Dog Constructor Called" << endl;
}

Dog::~Dog()
{
   cout << "Dog Destructor Called" << endl;
}
```

Notice that the constructor has the same name as the class. The destructor has the same name as the class prefixed by a tilde, "~". Above, the constructor was used to initialize member variables. In other classes, the constructor might allocate memory, acquire control of resources such as system devices, or perform more complicated initialization code. The destructor as defined above performs no real actions, other than echoing that it was called. In other classes, the destructor might free memory that was allocated, release some resources or perform some other clean up activity. As will be described in later lessons, it is possible to have multiple constructors that differ in their number and/or type of parameters. The constructor that is used is based on the arguments used in its invocation. This is referred to as function or method overloading.

Using Objects

To help understand the use of objects, the following program declares objects of the Dog class. For simplicity, all code will be contained in a single source file, although in larger projects classes are usually kept in separate files from the main program.

```
#include <iostream>
using std::cout;
using std::endl;

class Dog {
private:
   int age;
   int weight;
public:
   Dog();      //Constructor
   ~Dog();    //Destructor
   void setAge(int age);
   int getAge();
   void setWeight(int weight);
   int getWeight();
   void speak();
};

Dog::Dog()
{
   age = 0;
   weight = 0;
   cout << "Dog Constructor Called" << endl;
}

Dog::~Dog()
{
   cout << "Dog Destructor Called" << endl;
}
```

```
void Dog::setAge(int age)
{
   this->age = age;
}

int Dog::getAge()
{
   return age;
}

void Dog::setWeight(int weight)
{
   this->weight = weight;
}

int Dog::getWeight()
{
   return weight;
}

void Dog::speak()
{
   cout << "BARK!!" << endl;
}

int main()
{
   Dog fido;
   Dog rover;

   cout << "Rover is " << rover.getAge() << " years old." << endl;
   cout << "He weighs " << rover.getWeight() << " lbs." << endl;
   cout << endl;

   cout << "Updating Rover's Age and Weight" << endl;
   rover.setAge(1);
   rover.setWeight(10);

   cout << "Rover is " << rover.getAge() << " years old." << endl;
   cout << "He weighs " << rover.getWeight() << " lbs." << endl;
   cout << endl;

   cout << "Fido is " << fido.getAge() << " years old." << endl;
   cout << "He weighs " << fido.getWeight() << " lbs." << endl;

   cout << "Setting Fido to be the same as Rover" << endl;
   fido = rover;
```

```
   cout << "Fido is " << fido.getAge() << " years old." << endl;
   cout << "He weighs " << fido.getWeight() << " lbs." << endl;

   rover.speak();
   fido.speak();

   return 0;
}
```

The first two lines of the main program create or instantiate two objects, fido and rover, of class Dog. Rover's age and weight are printed and then updated using the accessor functions, setAge and setWeight. Notice that the methods are called for a particular instance of the class. That is, they are called as rover.method(). The dot operator, ".", is used to access a class member or method. Further down in the program fido is set equal to rover. This causes each member of rover to be copied into fido's members. As will be discussed in a later lesson, this member-wise copy may not be appropriate in all cases, particularly when the class contains pointers to other objects. Finally, as all dogs do, rover and fido decide to bark.

Object Terminology

This section provides definitions for three common terms encountered in object-oriented design and programming.

Encapsulation: The property of being self-contained. Encapsulation is really data hiding. Private data, public interface. The members of the class are private, the accessor methods are public. Programs using the class need not know the inner details of the class. If the inner workings of the class are changed but the interface remains constant then programs using the class do not need to be modified, only recompiled.

Inheritance: A subclass may be derived from a class and inherit its methods and members. The subclass will be more specialized. For instance, we could create a vehicle class that has members and methods that apply to all vehicles. For instance, all vehicles might have a member to store velocity and a "brake" function. A plane class derived from the vehicle class could add specialized features such as an altitude member or a landing method. Inheritance allows code to be developed without reinventing the wheel at each step.

Polymorphism: Polymorphism refers to the ability of an object to behave in different ways based on the context. The same function may exist in related classes. For instance both a general "Car" class and derived "Ford" and "Chevy" classes might have accelerate functions. A program using these classes might at run-time choose to use one of these classes and the correct accelerate function will be resolved as the program runs. A second form of polymorphism results from the overloading of functions. Multiple functions can have the same name but different arguments. The correct one is chosen based on the arguments used in its invocation.

Classes, Members and Methods

The use of objects in C++ defines the way programs are designed and written. Classes are a software construct that can be used to emulate a real world object. Any real world object can be defined by its attributes and by its actions. For instance, a cat has attributes such as its age, weight, and color. It also has a set of actions that it can perform such as sleep, eat and complain (meow). The class mechanism in C++ provides a way to create "cats" that we can use in our program. Obviously, no software construct can perfectly emulate a real object, but a goal in designing classes is to have all the relevant attributes and abilities encapsulated by the class. This way, objects of a class are easily created and used. In essence, a ready-made cat is also available to fulfill any requirements.

The difference between a class and an object can be confusing to beginners. A class is a programmer defined data type that encapsulates data and abilities. For instance, a software model of a car, a car "class", might contain data about the type of car and abilities such as accelerate or decelerate. An object is a particular instance of a class. This is best understood by an analogy with a built in data type such as int.

```
int x;
```

Declares x to be a variable of type int.

```
Car impala;
```

Declares impala to be an object of class Car.

Only an object represents an actually entity that can be manipulated, initialized or assigned. The class is a data type.

A number cannot be assigned to "int", since it is a data type and not a variable.

```
int = 52;    // Incorrect

int x;    // Correct, declares a variable, x, of built-in type int.
x = 52;
```

Similarly, the "class" generally should not appear in expressions. The exception to this will be seen later in a later lesson covering static members and methods, which are class wide.

```
Car = impala;    // Incorrect

Car myCar;    //Correct, defines a object, myCar, of user defined class Car.
myCar = impala;    //Assigns impala to myCar.
```

Defining Classes

A class is defined by using the keyword class followed by a programmer-specified name followed by the class definition in braces. The class definition contains the class members, its data, and the class methods, its functions. As an example, let's design an "Image" class that will be used to store and manipulate an image. Let's step back and consider what's needed to store an image and the types of manipulations that are needed. First, an image has a width and height measured in pixels. For instance, an image might be 400 pixels wide and 300 pixels high. Each pixel, each point in the image, has properties that define its color and intensity. One common scheme to represent a pixel is to assign and maintain separate red, blue and green intensities for each. These intensities will vary from 0 to 1. 0 will indicate that none of that color is at a particular pixel, while 1 will indicate the maximum intensity. So, we'll need data members to store image width and height, and the red-green-blue values of each pixel. What abilities or methods does our class need? First, let's assume that the height and width of the image are constrained to be less that 400 pixels and that these values will be set by the constructor on object creation. The constructor is a special method that is called at the creation of an object. Constructors will be covered in the next lesson. We won't need methods to set width and height, but we will want to get or read these values. Second, we need a way to set and get the color values for a particular pixel. This requires that we know our present position within the image. To know what pixel we're at, we will add members that record our current position in the file.

Here is the first version of the Image class we will develop in stages and use in the next few lessons.

```
class Image {
public:
   int getWidth();
   int getHeight();
   void setX(int x);
   int getX();
   void setY(int y);
   int getY();
   void setRed(double red);
   double getRed();
   void setBlue(double blue);
   double getBlue();
   void setGreen(double green);
   double getGreen();
private:
   int _width;
   int _height;
   int _x;
   int _y;
   double _red[400][400];
   double _blue[400][400];
   double _green[400][400];
   boolean isWithinSize(int s);
   double clipIntensity(double brightness);
```

```
};
```

Let's look at this class definition carefully. First, notice that there are two new keywords: private and public. These are called access specifiers. Private indicates that a member or method may be accessed only be class methods and not by other parts of the program or from other classes. The keyword public indicates that a member or method may be accessed other parts of a program using objects of this class. This technique of limiting access and manipulation of members only through methods is referred to as data hiding. The interface to the class is public and the data is private. For instance, a particular data member, say the position, _x, is not accessed directly from code outside of the scope of the class. It is accessed via the public methods, setX and getX. Public interface, private data is a key concept when designing classes. Good class design should always enforce data hiding. It is rarely necessary or desirable to directly access the internal data of a class.

Data hiding accomplishes two goals. First, users of the class need not be concerned with the internal representation of the data, that is, how it is stored. Whether the intensities are stored as three 2 dimensional arrays, as a single six dimensional array, as a vector of 3 tuples, or any of countless other ways need not be considered by a user of the image class. Second, if the internal representation of the data is modified, as long as the interface, that is the return types and argument lists of the public methods, remain unchanged, any code using this class need not be modified. Data hiding simplifies programming by hiding the inner workings of the class and protects the user from future changes to the class. Also notice that methods as well as members may be private to a class. Private methods are usually helper functions for the other methods of the class. They may be called from within the other methods of the class but not from outside of the class. If no access specifiers are used, all members and methods have private access by default. I recommend that you always specify the level of access. Finally, as a matter of style, I like to prefix private members with an underscore.

Side Note: C++ provides a third access specifier, protected. Protected is used to allow access to the internal data of a base class by derived classes. We'll study this more in the course: Tecnolgie Object Oriented.

Defining Methods

In the class definition on the last page, the methods are declared but not defined. That is, an implementation for each method must be written. There are two ways to define methods. They may be defined within the class definition, which is most appropriate for small methods. Larger methods may be defined outside of the class. In this case, they must be identified as belonging to that class. This is done with the scope resolution operator, "::". It identifies each method as belonging to a particular class. Methods defined within a class definition are automatically inlined. The compiler expands inlined functions or methods at their point of invocation. This means, that the compiler expands the code of inlined functions within object files at the point the functions are called. For a non-inlined function or method, execution jumps to the function or method as a program runs. Function variables are popped onto the stack, the function runs, and then function variables are popped off the stack (discarded). Inline functions avoid these steps, so they run faster. The downside is that the local expansion of the function or method consumes extra memory. The keyword inline may be used with methods defined outside the class

definition.

Below is the Image class with some methods defined.

```
class Image {
public:
   int getWidth() {return _width;}
   int getHeight();
   void setX(int x);
   int getX() {return _x;}
   void setY(int y);
   int getY();
   void setRed(double red);
   double getRed();
   void setBlue(double blue);
   double getBlue();
   void setGreen(double green);
   double getGreen();
private:
   int _width;
   int _height;
   int _x;
   int _y;
   double _red[400][400];
   double _blue[400][400];
   double _green[400][400];
   boolean isWithinSize(int s);
   double clipIntensity(double brightness);
};

//Notice use of scope resolution operator
void Image::setRed(double red)
{
   _red[_x][_y] = red;
}
```

Notice that methods have full access to all data members and can call any of the methods of the class. The access specifiers limit access from the outside only.

Practice
Complete the definitions for all methods in the above class. For now, don't worry about constructors, destructors and isWithinSize. They will be covered in the next lesson. Assume that the constructor sets the initial width and height, and that it uses the method isWithinSize to limit the width and height to 400 pixels. Hint: For setX and setY, check the size before you set.
Solution

```
class Image {
public:
   int getWidth() {return _width;}
   int getHeight() {return _height;}
   // Assumes valid width set by class constructor.
   // Covered in next lesson.
   void setX(int x)
   {
      if (x <= _width) {
         _x = x;
      }
      else {
         _x = _width;
      }
   }
   int getX() {return _x;}
   // Assumes valid height set by class constructor.
   // Covered in next lesson.
   void setY(int y)
   {
      if (y <= _height) {
         _y = y;
      }
      else
{
         _y = _height;
      }
   }
   int getY() {return _y;}
   void setRed(double red);
   double getRed() {return _red[_x][_y];}
   void setBlue(double blue);
   double getBlue() {return _blue[_x][_y];}
   void setGreen(double green);
   double getGreen() {return _green[_x][_y];}
private:
   int _width;
   int _height;
   int _x;
   int _y;
   double _red[400][400];
   double _blue[400][400];
   double _green[400][400];
   bool isWithinSize(int s);
   double clipIntensity(double brightness);
};

//Notice use of scope resolution operator
double Image::clipIntensity(double brightness)
{
   if (brightness > 1.0)
   {
      brightness = 1.0;
   }
   if (brightness < 0.0)
```

```
    {
        brightness = 0.0;
    }
    return (brightness);
}

//Notice use of scope resolution operator
void Image::setRed(double red)
{
    red = clipIntensity(red);
    _red[_x][_y] = red;
}

void Image::setBlue(double blue)
{
    blue = clipIntensity(blue);
    _blue[_x][_y] = blue;
}

void Image::setGreen(double green)
{
    green = clipIntensity(green);
    _green[_x][_y] = green;
}
```

Overloaded Methods and Default Arguments

The methods of a class are functions that are part of a class. They share properties with ordinary functions. They can be overloaded. Function overloading provides a way to have multiple functions with the same name.

For an example, let's overload the setRed function in our Image class. Assume that it will either set the red intensity at the pixel at the current positions of _x and _y, or allow the red intensity, and the current position (_x and _y) to be set.

Here is the Image class showing only the modifications to overload the setRed function.

```
class Image {
public:
...
    void setRed(double red);
    void setRed(double red, int x, int y);
...
private:
....
};


//Notice use of scope resolution operator
void Image::setRed(double red)
{
    red = clipIntensity(red);
```

```
    _red[_x][_y] = red;
}

void Image::setRed(double red, int x, int y)
{
    if (y <= _height) {
        _y = y;
    }
    else {
        _y = _height;
    }
    if (x <= _height) {
        _x = x;
    }
    else {
        _x = _height;
    }

    red = clipIntensity(red);
    _red[_x][_y] = red;
}
```

As with ordinary functions it is also possible to specify default arguments for methods in a class. Any defaults must be specified starting at the right most parameters. That is, a default for a particular parameter can be specified only if all parameters to the right of it in the parameter list also have defaults. This is best understood with a simple example.

```
void someFunction(int arg1, int arg2, int arg3, int arg4);    // Valid
void someFunction(int arg1, int arg2, int arg3, int arg4 = 0);    // Valid
void someFunction(int arg1, int arg2, int arg3 = 52, int arg4 = 0);    // Valid
void someFunction(int arg1, int arg2, int arg3 = 52, int arg4);    //Not Valid
```

As with ordinary functions, it is possible to specify the default arguments in either the declaration or the definition of the method. I recommend that it be done in the method declaration, that is, within the class definition. The reason for this is that a programmer using the class will need only look at the class definition to see the defaults and need not look at the actual implementation. The class definitions are typically within include files. The method definitions will typically be in separate source files. Users of a class should only need to examine include files to use it.

As an example, lets provide a default value for the setX method of our Image class. A reasonable default is to set _x to 0 if no value is provide. Here is the class showing only this change.

```
class Image {
```

```
public:
...
   void setX(int x = 0)
...
private:
...
};
```

No other change is required to specify this default.

Practice: Try expanding your Image class to provide overloaded versions of setBlue and setGreen. Provide a default argument for setY.

Classes As Members

The members of a class may be any built-in or user-defined data types, including other classes. A class may be contained by value within another class only if it is already defined. That is, the class definition of the contained class must appear in the source file prior to the definition of the containing class. The definition of the contained class could be within the source file directly or be included in an "include" file. Here is a simple example. I want to define a Car class that will have an Engine class and a Tire class as components.

```
class Engine {
public:
... //Accessor Methods
private:
   int numValues;
   double displacement;
...
};

class Tires {
public:
... //Accessor Methods
private:
   int size;
   double pressure;
};

class Car {
public:
... //Accessor Methods
private:
   int numDoors;
   Engine _myEngine;
   Tires _myTires[4];
};
```

If a class is declared but not defined in a file, a pointer or reference to it can be made within another class, but it cannot be included by value. The reason for this is that the compiler does not know the size of the contained class. Pointers and references are of fixed size; large enough to contain an address, so the exact size of the contained class is not required.

```
class Engine; // Declares that a class exists named Engine;
class Tires { // Defines the Tires class
public:
... //Accessor Methods
private:
   int size;
   double pressure;
};

class Car {
public:
... //Accessor Methods
private:
   int numDoors;
   Engine *myEngine;
   Tires _myTires[4];
};
```

Being able to declare references and pointers to classes that are declared, but not defined is particularly useful. Several important data structures can be implemented using this ability. In particular, it allows an object to access other objects of the same class. The classic example is a linked list.

```
class Node {
... // Definition of a node to contain some type of data
...
}
class Link {
public:
... //Accessor methods
private:
   Node _node;
   Link *next;
   Link *prev;
};
```

Constructors and Destructors

Constructors and destructors are special class methods. A constructor is called whenever an object is defined or dynamically allocated using the "new" operator. The purpose of a constructor is to initialize data members and sometimes to obtain resources such as memory, or a mutex or lock on a shared resource such as a hardware device. An object's destructor is called whenever an object goes out of scope or when the "delete" operator is called on a pointer to the object. The purpose of the destructor is clean up. It is used to free memory and to release any locks or mutexes on system resources.

The definition and use of constructors and destructors is fairly simple. First, I'll outline some basic rules on constructors and destructors, and then provide the details using some simple examples.

- A constructor is a method that has the same name as its class.
- A destructor is a method that has as its name the class name prefixed by a tilde, ~.
- Neither constructors nor destructors return values. They have no return type specified.
- Constructors can have arguments.
- Constructors can be overloaded.
- If any constructor is written for the class, the compiler will not generate a default constructor.
- The default constructor is a constructor with no arguments, or a constructor that provides defaults for all arguments.
- The container classes such as vector require default constructors to be available for the classes they hold. Dynamically allocated class arrays also require a default constructor. If any constructors are defined, you should always define a default constructor as well.
- Destructors have no arguments and thus cannot be overloaded.

Let's assume that we're creating a class that will hold employee information such as salary and ID number. Here's the first version of our class along with a program showing its use.

```
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Employee {
public:
   int getID() {return _id;}
   void setID(int id) {_id = id;}
   double getSal() {return _sal;}
   void setSal(double sal) {_sal = sal;}
private:
   int _id;
   double _sal;
};
```

```
int main()
{
   Employee john;
   john.setID(1);
   john.setSal(10.0);

   cout << "Employee " << john.getID() << " is vastly underpaid at" << john.getSal() <<
endl;

   return 0;
}
```

OUTPUT:
Employee 1 is vastly underpaid at 10.0

An astute reader may notice two things. First, the author is under compensated. Second, and more significant, there are no constructors and destructors in the class definition. The compiler creates a default constructor and destructor for a class if none are provided. These default methods do nothing. Data members are not initialized. The class can still be used and accessor methods can set variables, but this is cumbersome and error prone. It is easy to neglect to initialize a variable. A better programming practice is to always define constructors and destructors.

The constructor for our Employee class must serve the same purpose as all constructors. It must initialize the data members of the class. Here is an example of a default constructor for this class. It takes no arguments.

```
Employee::Employee()
{
   _id = 0;
   _sal = 0.0;
}
```

Let's write a constructor that allows the initial values to be specified.

```
Employee::Employee(int id, double sal)
{
   _id = id;
   _sal = sal;
}
```

Constructors may be overloaded. There is no limit on the number of constructors that can be written for a class, provided that each differs in the number or type of its arguments. Here is an example that allows the id to be specified while initializing the salary to zero.

```
Employee::Employee(int id)
{
   _id = id;
   _sal = 0.0;
}
```

Finally, let's try rewriting the default constructor. A default constructor can have parameters provided that every parameter has a default value.

```
Employee::Employee(int id = 0, double sal = 0.0)
{
   _id = id;
   _sal = sal;
}
```

Notice that this last constructor actually can replace all of the constructors before it. It works correctly with no arguments, with just "id" specified, or with "id" and "sal" specified.

Now lets look at a destructor for this class. Destructors are used to free memory, release resources and to perform other clean up. For this simple class with all its members held by value, a destructor really serves no purpose. We will see examples in latter lessons where a destructor is absolutely required. For now, let's just define an empty destructor. Notice that a destructor takes no arguments and returns no value.

```
Employee::~Employee()
{
}
```

Let's put all this together in a program to better understand how it works.

```
#include <iostream>
using std::cout;
using std::endl;

class Employee {
public:
   Employee(int id = 0, double sal = 0.0);
   int getID() {return _id;}
   void setID(int id) {_id = id;}
   double getSal() {return _sal;}
   void setSal(double sal) {_sal = sal;}
   ~Employee() {} // This could be defined outside
                  // the class definition, instead
private:
   int _id;
```

```
    double _sal;
};

// This could be defined within the class definition, instead.
Employee::Employee(int id, double sal)
{
    _id = id;
    _sal = sal;
}

int main()
{
    Employee manager(1,3000.0);
    Employee janitor(2,300.0);
    Employee programmer(3,2000.0);

    cout << "Employee " << manager.getID() << " earns " << manager.getSal() << endl;
    cout << "Employee " << programmer.getID() << " earns " << programmer.getSal() <<
endl;
    cout << "Employee " << janitor.getID() << " earns " << janitor.getSal() << endl;
    cout << "Don't be a fool, stay in school" << endl;
    cout << "Don't start forest fires" << endl;

    //Demotion
    manager.setSal(300.0);

    //Promotion
    janitor.setSal(3000.0);

    cout << "Employee " << manager.getID() << " earns " << manager.getSal() << endl;
    cout << "Employee " << programmer.getID() << " earns " << programmer.getSal() <<
endl;
    cout << "Employee " << janitor.getID() << " earns " << janitor.getSal() << endl;

    return 0;
}
```

Let's extend our class to allow more dignity for our workers. We'll allow them to be identified by name as well as by ID number. :-). A new data member, _name of type string will be added to the Employee class.

```
class Employee {
public:
    Employee(string name, int id = 0, double sal = 0.0);
    Employee()
    {
        //default constructor for string, _name, implicitly called
        _id = 0;
```

```
        _sal = 0.0;
    }
    int getID() {return _id;}
    void setID(int id) {_id = id;}
    double getSal() {return _sal;}
    void setSal(double sal) {_sal = sal;}
    string getName() {return _name;}
    void setName(string name) {_name = name;}
    ~Employee() {} // This could be defined outside
                   // the class definition instead
private:
    int _id;
    double _sal;
    string _name;
};

// This could be defined within the class definition instead.
Employee::Employee(string name, int id, double sal) : _name(name)
{
    _id = id;
    _sal = sal;
}
```

There are a few things to notice in this example. First, look at the constructor that is defined outside of the class. The "_name" member is initialized as part of a member initialization list. The member initialization list provides an alternate syntax for initializing members of a class within a constructor. It is a comma-separated list of members along with their initial values, separated by a colon from the end of the parameter list and before the opening bracket of the constructor body. All data members could be specified in the initialization list. In that case, the constructor would be as follows.

```
Employee::Employee(string name, int id, double sal) : _name(name), id(id), _sal(sal)
{
}
```

A common question is whether data members should be initialized within a member initialization list or within the body of the constructor. For members of intrinsic data types such as id and sal, it is a matter of preference. For members that are themselves classes, they should always be initialized in the member initialization list. The only way to initialize a member that is a class, that is, the only way to call a particular form of the member class's constructor with some arguments is via the initialization list. With the above Employee constructor, the string class's constructor is called with "name" as an argument. Suppose we try to initialize "_name" within the constructor body.

```
Employee::Employee(string name, int id, double sal)
```

```
{
   _name = name;
   _id = id;
   _sal = sal;
}
```

This is also legal C++ code, but there is a subtle difference. Here, the default string constructor is called to create "_name", and then the value "name" is assigned to the string variable "_name". For strings, the net result is the same either way. For other classes, it may not be. Moral: always initialize classes within a member initialization list. This way you control how the class member is initialized. Within the body of the constructor you invoke the default constructor followed by assignment.

There are two other cases where members must be initialized within a member initialization list. Members that are either references or "const" must also be initialized within an initialization list.

The best way to learn is by doing, so here is a practice problem. Write a program that defines and the uses a "patient" class. This class should have the following data members.

1) patient name
2) patient heart rate
3) amount of money owed by patient to doctor.

Steps:

1) Decide on the data types required to store the above data fields.
2) Write a Patient class with appropriate accessor methods, constructors and destructors.
3) Write a small program to exercise this class.

Solution

```
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Patient {
public:
   Patient(string name, int heartRate = 0, double moneyOwed = 0.0);

   Patient()
   {
      //default constructor for string, _name, implicitly called
      _heartRate = 0; //Bad for business
      _moneyOwed = 0.0;
   }

   int getHeartRate() {return _heartRate;}
   void setHeartRate(int heartRate) {_heartRate = heartRate;}
```

```
    double getMoneyOwed() {return _moneyOwed;}
    void setMoneyOwed(double moneyOwed) {_moneyOwed = moneyOwed;}
    string getName() {return _name;}
    void setName(string name) {_name = name;}
    ~Patient() {} // This could be defined outside
                  // the class definition instead

private:
    int _heartRate;
    double _moneyOwed;
    string _name;
};

// This could be defined within the class definition instead.
// Always initialize member classes within the member initialization list.
Patient::Patient(string name, int heartRate, double moneyOwed) : _name(name)
{
    _heartRate = heartRate;
    _moneyOwed = moneyOwed;
}

int main()
{

    Patient jk("John",70,0.0);

    //Before visit to doctor
    cout << "Patient " << jk.getName() << " owes " << jk.getMoneyOwed() << endl;
    cout << "His heart rate is " << jk.getHeartRate() << endl;

    //After visit to doctor
    jk.setMoneyOwed(1000.0);
    jk.setHeartRate(140);
    cout << "Patient " << jk.getName() << " owes " << jk.getMoneyOwed() << endl;
    cout << "His heart rate is " << jk.getHeartRate() << endl;

    return 0;
}
```

Copy Constructors

A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy. By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created. This is called a member wise or shallow copy. Although it may seem to be the desired behavior, in many cases a shallow copy

is not satisfactory. To see why let's look at the Employee class introduced in an earlier lesson with one change. We will store the name in a C-style character string rather than store the employee name using the string class from the standard C++ library. Here is a simple program with a bare bones version of the Employee class.

```
#include <iostream>
using std::cout;
using std::endl;

class Employee {
public:
   Employee(char *name, int id);
   ~Employee();
   char *getName(){return _name;}
   //Other Accessor methods
private:
   int _id;
   char *_name;
};

Employee::Employee(char *name, int id)
{
   _id = id;

   _name = new char[strlen(name) + 1];
         //Allocates an character array object
   strcpy(_name, name);
}

Employee::~Employee()
{
   delete[] _name;
}

int main()
{
   Employee programmer("John",22);
   cout << programmer.getName() << endl;
   return 0;
}
```

The function strlen returns the length of the string passed into the constructor. Notice that the Employee name is now stored in a dynamically allocated character array. It is of "string length + 1" to allow for the null terminator used with C-style strings, '\0'. The strcpy function automatically adds the null terminator to the destination string. Also, notice that the destructor frees the memory used to hold the employee name. This is needed to avoid a memory leak.

Now suppose that "John" is promoted.

```
int main()
{
   Employee programmer("John",22);
   cout << programmer.getName() << endl;

   //Lots of code ....

   Employee manager(&programmer);
      //Creates a new Employee "manager",
      //which is an exact copy of the
      //Employee "programmer".

   return 0;
}
```

The above program contains a serious bug and will die with an exception when run. The problem is that the default member-wise copy constructor is being used to create the "manager" object. This provides correct behavior for members held by value, such as _id. We want the ID number copied from "programmer" to "manager". But the member-wise copy copies the address stored in the pointer _name in "programmer" to the pointer _name in "manager". We now have two pointers both containing the same address, that is, both referring to the same block of memory in the free store. Suppose that a new individual is hired as "programmer". When the name is updated, it will not only change the name of the programmer, but also the manager! Remember, there is only a single block in the free store and both objects hold pointers to it. This is surely not what we want. Additionally, when the first of these objects goes out of scope its destructor will be called. The block of memory in the free store will be released. When the second object tries to reference this memory, bad things happen. For instance, when I run the above program, programmer's destructor gets called first. It works fine and frees the block in memory. When manager's destructor is called, an exception is thrown when the delete is attempted because that block of memory has already been released.

The solution to this problem is to define our own copy constructor. When a new object is created using this constructor, we need to be sure to allocate a new block of memory in the free store and to copy the name itself rather than the pointer to name. By convention, the object passed into copy constructors is called "rhs", for right hand side. Here is the Employee class with a copy constructor defined.

```
class Employee {
public:
   Employee(char *name, int id);
   Employee(Employee &rhs);
   ~Employee();
   char *getName(){return _name;}
   int getId() {return _id;}
   //Other Accessor methods
private:
```

```
    int _id;
    char *_name;
};

Employee::Employee(char *name, int id)
{
    _id = id;

    _name = new char[strlen(name) + 1];
          //Allocates an character array object
    strcpy(_name, name);
}

Employee::~Employee() {
    delete[] _name;
}

Employee::Employee(Employee &rhs)
{
    _id = rhs.getId();
    _name = new char[strlen(rhs.getName()) + 1];
    strcpy(_name,rhs._name);
}
```

As a concluding note, the reference passed into the copy constructor should be declared const.

Arrays of Class Objects

Review: Defining and Using Class Objects
The simplest way to learn how to define and use classes is to study a simple example. Let's define a simple Cat class, create a few felines and see what they can do. But, first, let's quickly look at dynamically allocating class objects. We saw the use of new and delete with built-in types in a previous lesson. The operator new dynamically creates an object on the heap and returns a pointer to it. The operator delete is used to free that memory after it is no longer needed so that it may be reused elsewhere by the program. The following syntax is used to dynamically allocate a class object.

```
class pt = new class;    //Calls default constructor
class pt = new class(arg1, arg2, ....);    /* Calls constructor with a parameter list that matches the number and type of the arguments in the call. */
```

To delete the dynamically allocated object, that is, to free the memory it occupies:

```
delete pt;
   //Where pointer contains the address of the object to be freed.
```

Hiss!!. Here is a brief example.

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;

class Cat {
public:
   Cat(string name = "tom", string color = "black_and_white") : _name(name), _color(color)
{}
   ~Cat() {}
   void setName(string name) {_name = name;}
   string getName() {return _name;}
   void setColor(string color) {_color = color;}
   string getColor() {return _color;}
   void speak() {cout << "meow" << endl;}
private:
   string _name;
   string _color;
};

int main()
{
   Cat cat1("morris","orange");    //Objects of automatic extent exist on stack.
   Cat *cat2pt = new Cat("felix","black");    /* Dynamically allocated objects exist on the
heap. */
   Cat *cat3pt = new Cat;    //Calls default constructor

   cout << cat1.getName() << " is " << cat1.getColor() << endl;
   cout << cat2pt->getName() << " is " << cat2pt->getColor() << endl;
   cout << cat3pt->getName() << " is " << cat3pt->getColor() << endl;

   cat1.speak();
   cat2pt->speak();

   delete cat2pt;    //Always "delete" dynamically allocated objects!
   delete cat3pt;

   return 0;
}
```

There are several important things to note in this example.

Methods of automatic objects are accessed using the "." operator. For example, cat1.getName().

Methods of dynamically allocated objects are accessed using the "->" operator. For example, cat2pt->getName().

A default constructor can have either no arguments or provide defaults for all arguments.

If no arguments are specified when dynamically allocating an object, the default constructor is called.

It is the programmers responsibility to delete dynamically allocated objects when they are no longer needed. Otherwise, a memory leak is created.

Arrays of Class Objects

An array of class objects is defined in the same manner as build-in types.

```
Cat myCats[27];
```

Defines an array to hold 27 Cat objects, one for each cat I own. Which constructor is used? The default constructor is used because no arguments are specified. The are several ways to use other constructors to form these objects, that is, to pass arguments to the constructor. An array of intrinsic type can be specified using an array initialization list. For instance,

```
int primes[5] = {1,2,3,5,7};
```

defines an integer array with the values of the first five prime numbers. Likewise, for an array of class objects:

```
Cat myCats[4] = {"Chris","Charles","Cindy","Cathy"};
```

Notice that only the first argument to the constructor can be specified with this technique. To specify multiple arguments for each Cat object, an alternate syntax allows a constructor to be specified within the array initialization list.

```
Cat myCats[4] = {
   Cat("Chris","Gray"),
   Cat("Charles","White"),
   Cat("Cindy","BlueGray"),
   Cat("Cathy","Brown")
};
```

The syntax of array initialization lists is cumbersome. In practice, it is easier to define an array of class objects without an initialization list (using the default constructor) and then to use accessor methods to set the values of members. We'll see this later in an example.

A array of class objects is dynamically allocated exactly like an array of objects of built-in type.

```
int *catIDs = new int[27];
Cat *myCats = new Cat[27];
```

Notice here, in both cases, within the brackets is the number of objects to be allocated. The syntax does not allow for specification of initial values. The default constructor will be used. But, this is not much of a limitation. The initial values may still be set using the accessor methods of the class.

Here is how to delete an array of dynamically allocated class objects. The brackets are required. Compare this to the examples on the previous page that deletes single objects. Notice, in those cases, no brackets were used

```
delete [] catIDs;
delete [] myCats;
```

Here is a simple example program demonstrating the definition and use of arrays of class objects.

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;

class Cat {
public:
   Cat(string name = "tom", string color = "black_and_white") : _name(name), _color(color)
{}
   ~Cat() {}
   void setName(string name) {_name = name;}
   string getName() {return _name;}
   void setColor(string color) {_color = color;}
   string getColor() {return _color;}
   void speak() {cout << "meow" << endl;}
private:
   string _name;
   string _color;
};

int main()
{
   // Single Argument form of array initialization list
   Cat myCats[4] = {"Homer","Marge","Bart","Lisa"};

   // Emplicit use of constructors in array initialization list
   Cat myOtherCats[3] = {
```

```
    Cat("Chris","Gray"),
    Cat("Charles","White"),
    Cat("Cindy","BlueGray")
  };

  Cat *catpt = new Cat[27];
  Cat *pt;

  // Use accessor methods to initialize dynamically allocated class objects
  pt = catpt;
  for (int i = 0; i < 27; i++) {
    pt->setName("Felix");    // You can never have enough
    pt->setColor("Black");    // black cats named Felix
    pt++;    // Can also increment pointer within the for statement
  }

  // Verify number of Felix's
  for (int i = 0; i < 27; i++) {
    cout << (i+1) << ": " << catpt[i].getName() << endl;
    //Use of array notation with a pointer
  }

  myCats[0].speak();
  myCats[1].speak();

  delete[] catpt;    //Remember the braces when freeing the memory of an array

  return 0;
}
```

Practice
Try creating arrays of objects as shown in this lesson using the Cat class or any class of your choosing.

Const provides a way to declare that an object is not modifiable. It can also be used with class methods to indicate to they will not modify their objects. The use of "const" can reduce bugs within your code by allowing the compiler to catch unintended modification of objects that should remain constant. The keyword mutable provides a way to allow modifications of a particular data members of a constant objects. By the way, I love tabloid style headlines, which is why this lesson is titled "Mutable Members" rather than something dull like "Const and Mutable"

Const
In an earlier lesson, the keyword const was used to declare that an object of built-in type, that is, an ordinary variable, was a constant. Attempts to modify a "const" result in a compilation error.

```
const int limit;
limit = 25;    // Results in compilation error
```

Assigning to a const variable is not permitted. It must be initialized to provide a value.

```
const int limit = 25;
```

This provided a way to have a constant and to be sure that the constant was not modified unintentionally. C++ also allows the declaration of const class objects.

```
// Class Definition
class Employee {
public:
   string getName(){return _name;}
   void setName(string name) {_name = name;}
   string getid(){return _id;}
   void setid(string id) {_id = id;}
private:
   string _name;
   string _id;
};

const Employee john;
```

This declares the object john of class Employee to be constant. But there are problems with this code. First, since the object is const, we need a way to initialize it. Its members cannot be assigned either directly or indirectly via methods. The compiler's default constructor is insufficient, so we must define constructors that can initialize all the data members. A default constructor that initializes all members is required. Other constructors may be written. Second, C++ allows methods to be declared as const. By declaring a method "const" we are in effect stating that the method cannot change its object. *Only methods declared const can be called be a const object.* This has real benefit. The compiler can check that methods declared const do not modify the object. Attempts to modify an object within a const method will be flagged by the compiler. Since a const object may invoke only const methods, it will not be modified unintentionally. *Objects that are not const can invoke both const and non-const methods.* Which methods should be declared as const? Certainly, any method that is intended to simple return the value of a data member should be. Depending upon their purpose, the const keyword may be appropriate for other methods as well. Let's correct the Employee class to allow its proper use with const Employee objects and see its use in a simple program.

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string

class Employee {
```

```
public:
   Employee(string name = "No Name", string id = "000-00-0000") : _name(name), _id(id)
{}
   string getName() const {return _name;}
   void setName(string name) {_name = name;}
   string getid() const {return _id;}
   void setid(string id) {_id = id;}
private:
   string _name;
   string _id;
};

int main()
{
   const Employee john("John","007");

   // Const methods can be invoked by const objects
   cout << john.getName() << endl;
   cout << john.getid() << endl;

   // Non-const methods cannot be invoked by const objects
   john.setid("008");    //Results in compilation error.

   return 0;
}
```

Note that const methods can serve a purpose even if no const objects will be declared. By declaring that a method should not modify an object, many bugs can be avoided. For instance, suppose we have the following method that returns the age of an Employee. It contains a not so subtle bug. Since we are "getting" the age and not "setting", this method should be declared const.

```
int getAge() const
{
   _age = 1;    //Logic Error
   return _age;
}
```

Since the method is declared const, the compiler catches this bug. If the method were not const, the compiler would not catch this bug.

As a final note on const, if a method is declared outside of the class definition, the keyword const must be used in both its declaration within the class body and its definition.

```
class Employee {
public:
```

```
    int getAge() const;
    ....
private:
    ....
}

Employee::getAge() const
{
    return _age;
}
```

Static Members and Methods

Static members provide a way to create objects that are shared by an entire class, rather than being part of a particular instance of the class, an object. Static methods provide a way to access and update static members.

Static Members
Suppose it's necessary to keep track of some data that's related to a class as a whole, rather than to a particular object or instance of that class. A simple example is a count of the total number of objects instantiated. Maybe we have an Employee class and we want to keep track of the total number of employees. We could create a global variable to hold this data.

```
class Employee
{
    ....
}

int numEmployees;

main()
{
    Employee janitor;
    numEmployees++;

    Employee manager;
    numEmployees++;

    .....
}
```

While this approach will work, it has a major short fall. The global variable, numEmployees is not really related to the class Employee. Ideally, we want all data associated with the class to be encapsulated within that class. Although in this simple example it is obvious that numEmployees pertains to the Employee class, in larger

programs the relationship might not apparent. The static keyword provides a way to declare that an object is to be shared by all instances of a class. Here's an example.

Static Members, Example Program

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Employee {
public:
   Employee(string name = "No Name", string id = "000") : _name(name), _id(id)
   {
      numEmployees++;
   }
   ~Employee()
   {
      numEmployees--;
   }
   string getName() const {return _name;}
   void setName(string name) {_name = name;}
   string getid() const {return _id;}
   void setid(string id) {_id = id;}
   static int numEmployees;
private:
   string _name;
   string _id;
};

int Employee::numEmployees = 0;

main()
{
   Employee janitor("Tidy","123");

   cout << "Count: " << Employee::numEmployees << endl;
   cout << "Count: " << janitor.numEmployees << endl;

   Employee programmer("Messy","123");

   cout << "Count: " << Employee::numEmployees << endl;
   cout << "Count: " << programmer.numEmployees << endl;

}
```

Output:
Count: 1
Count: 1
Count: 2
Count: 2

There are several important things to notice in the previous example.

- Although the static member is declared within the class, it is defined outside of the class. A definition allocates space for a variable; a declaration simply gives the compiler information about its type. This makes sense because a class definition does not allocate space for the class. Space is allocated when an object is instantiated (defined). Since all objects of a class share a static data member, it does not belong within a particular object. It must be defined outside of any object.
- There are two ways to access a publicly declared static member. It can be accessed via the class name (Employee, in our example) or via any object (janitor or programmer) of the class.
- When accessing a static member via the class name, the scope resolution operator is used.
  cout << Employee::numEmployees;
- When accessing a static member via a particular object, the "dot" operator is used.
  cout << programmer.numEmployees;
- When accessing a static member via a pointer to a particular object, the "arrow" operator is used.
  Employee *ept;
  ept = &programmer;
  cout << ept->numEmployees;

Static Methods
In the previous section, we saw a static member declared with public access. It was accessible to the program at large, outside of any object of the class. This works, but it violates a goal of good object oriented design, data hiding. The data of a class should be accessed through the methods of a class, not directly. This limits incorrect modification and protects the programmer from changes in the way data is stored. If the members are accessed only through methods, then how they are stored is not of concern to anyone except the class designer. As long as the interface, that is, the accessor methods continue to return the same types, the internal data representation may be modified without changing code outside the class. Remember, private data, public interface. Static methods provide a way to access and update static members. An example of the use of static methods follows.

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Employee {
public:
```

```
   Employee(string name = "No Name", string id = "000-00-0000") : _name(name), _id(id)
   {
      numEmployees++;
   }
   ~Employee()
   {
      numEmployees--;
   }
   string getName() const {return _name;}
   void setName(string name) {_name = name;}
   string getid() const {return _id;}
   void setid(string id) {_id = id;}
   static int getNumEmployees() {return numEmployees;}
private:
   string _name;
   string _id;
   static int numEmployees;
};

int Employee::numEmployees = 0;

int main()
{
   Employee janitor("Tidy","123");

   cout << "Count: " << Employee::getNumEmployees() << endl;
   cout << "Count: " << janitor.getNumEmployees() << endl;

   Employee programmer("Messy","123");
   Employee *ept = &programmer;
   cout << "Count: " << Employee::getNumEmployees() << endl;
   cout << "Count: " << programmer.getNumEmployees() << endl;
   cout << "Count: " << ept->getNumEmployees() << endl;

   return 0; }
```

Notice that when a static method, such as getNumEmployees is invoked on a class, the scope resolution operator, ::, is used. When a static method is invoked by an object, an instance of a class, the dot, ".", operator is used. When a static method is invoked through a pointer, the arrow, "->", operator is used.

As a concluding point, static methods can only access static members, not any other class members. This makes sense. It would not be possible to determine which non-static members, that is, which object's members to access when the static method was called on a class. Non-static methods can access both static and non-static members. This is also reasonable. Since these methods are called on a particular

object, its members as well as static members that belong to the class can be uniquely distinguished.

This Pointer

This lesson covers the purpose and use of the "this" pointer. In addition to the explicit parameters in their argument lists, every class member function (method) receives an additional hidden parameter, the "this" pointer. The "this" pointer addresses the object on which the method was called. There are several cases when the "this" pointer is absolutely required to implement desired functionality within class methods.

Purpose
To understand why the "this" pointer is necessary and useful let's review how class members and methods are stored. Each object maintains its own set of data members, but all objects of a class share a single set of methods. This is, a single copy of each method exists within the machine code that is the output of compilation and linking. A natural question is then if only a single copy of each method exists, and its used by multiple objects, how are the proper data members accessed and updated. The compiler uses the "this" pointer to internally reference the data members of a particular object. Suppose, we have an Employee class that contains a salary member and a setSalary method to update it. Now, suppose that two Employees are instantiated.

```
class Employee {
public:
   ....
   void setSalary(double sal)
   {
      salary = sal;
   }
private:
   ....
   double salary;
   ....
}

int main()
{
   ....
   Employee programmer;
   Employee janitor;

   janitor.setSalary(60000.0);
   programmer.setSalary(40000.0);
   ....
}
```

If only one setSalary method exists within the binary that is running, how is the correct Employee's salary updated? The compiler uses the "this" pointer to correctly identify the object and its members. During compilation, the compiler inserts code for the "this" pointer into the function. The setSalary method that actually runs is similar to the following pseudo-code.

```
void setSalary(Employee *this, float sal)
{
   this->salary = sal;
}
```

The correct object is identified via the "this" pointer. So what, you say. Interesting, maybe. But if this implicit use of the "this" pointer were all there was, this topic would mainly be of interest to compiler designers. As we will see, explicit use of the "this" pointer also has great utility.

Concatenating Calls

Explicit use of the "this" pointer allows the concatenation of calls on an object. Suppose we have a class used to represent a point in 2D space. We will store the x and y coordinates and have several methods to manipulate the point. For simplicity, we will only code setX, setY and doubleMe methods. The doubleMe method will double both the X and Y coordinates. Here's a first version of the class.

```
class Point {
public:
   void setX(int x) {_x = x;}
   void setY(int y) {_y = y;}
   void doubleMe()
   {
      _x *= 2;
      _y *= 2;
   }
private:
   int _x;
   int _y; };
```

To use this class, we would code something like this.

```
Point lower;
lower.setX(20);
lower.setY(30);
lower.doubleMe();
```

A more natural way to handle this series of operations on a single point object would be to concatenate the calls.

```
lower.setX(20).setY(30).doubleMe();
```

How can we get this to work? The member access operators, dot, ".", and arrow, "->", are left associative. That means that the above expression is evaluated from left to right.

```
((lower.setX(20)).setY(30)).doubleMe();
```

That is, "setY(30)" is called on the result of lower.setX(20). If lower.setX(20) returned an object of the class point rather than void, we would have what we need. Likewise, if (lower.setX(20)).setY(30) returned an object of class point as well, then doubleMe() could be called on the result. This can be accomplished with the "this" pointer. We code each method to return a reference to an object of the class type. We return a reference rather than a copy for efficiency. Here is an updated version of the Point class that supports concatenation of calls.

```
class Point {
public:
   Point& setX(int x)
   {
      _x = x;
      return *this;
   }
   Point& setY(int y)
    {
      _y = y;
      return *this;
   }
   Point& doubleMe()
   {
      _x *= 2;
      _y *= 2;
      return *this;
   }
private:
   int _x;
   int _y;
};
```

Resolving Ambiguities

The "this" pointer is also useful if it is desired to use the same identifier for both a local variable within a method and for a class member. Let's look at the class in the last example, Point. Suppose that rather than use "_x" and "_y", the members used to represent the coordinates of the "point" are "x" and "y". Further suppose that we want to use the identifiers "x" and "y" in the parameter list of the methods setX()

and setY(). This is not unreasonable. After all, there is only a single x coordinate and a single y coordinate. Why complicate things with multiple identifiers. Within the methods, the local x and y will mask (override) the "x" and "y" class members. The "this" pointer can be used to access the class members in this case.

```
class Point {
public:
   Point& setX(int x)
   {
      this->x = x;
      // The "this" pointer allows access to the class member
      return *this;
   }
   Point& setY(int y)
    {
      this->y = y;
      return *this;
   }
   Point& doubleMe()
   {
      x *= 2;
      y *= 2;
      // x and y refer to the class members.
      return *this;
   }
private:
   int x;
   int y;
};
```

Notice that the "this" pointer is not required within the doubleMe method. There is no ambiguity.

Self Identification

When implementing some methods, it is important to check for identity. The classic example is a copy assignment operator for a class that has some dynamically allocated members. As will be seen in a later lesson, C++ allows operators such as +, =, -, among others, to be overloaded. For example, "+" has meaning with respect to built-in types such as int and float, but an addition operation may also be defined for classes. Exactly what it may mean to add two objects is very specific to the class, but C++ allows the "+" operator to be overloaded for this use. For example, using the employee class defined earlier in this lesson:

```
Employee janitor;
Employee programmer;
Employee manager;

manager = janitor + programmer;
```

//The "+" operation must be user defined, as we will see in a later lesson.

Likewise, the assignment operation, "=", may also be overloaded. For now, don't worry about the syntax or semantics of overloading. We're concerned only with the use of the "this" pointer.

```
class Employee {
public:
   Employee(char employeename[]);
   Employee& operator=(const Employee &);
   ....
private:
   char *name;
};
Employee::Employee(char employeename[])
{
   name = new char[sizeof(employeename) + 1];
   strcpy(name,employeename);
}
Employee& Employee::operator=(const Employee& rhs)
{
   delete [] name;
   name = new char[sizeof(rhs.name) + 1];
   strcpy(name,rhs.name);

   return *this;
}

int main()
{
   .....
   Employee a("John");
   Employee b("Janitor");
   ....
   a = b;
.....
}
```

There's a very subtle problem with this implementation. It's fine the way it's used in the above program, but what if the following error is made.

```
a = a;
```

That is, an Employee object is assigned to itself. This is unlikely to occur directly, but if the objects were stored in an array or list or referenced by pointers it could happen. The first step in the assignment operator deletes the name buffer. The sizeof

operation in the next line will bomb. "rhs.name" has been deleted. The program will die. A simple check for identity using the "this" pointer will prevent this.

```
Employee& Employee::operator=(const Employee& rhs)
{
   if (this != &rhs)
   {
      delete [] name;
      name = new char[sizeof(rhs.name) + 1];
      strcpy(name,rhs.name);
   }

   return *this; }
```

As a final note, the above implementation also does a deep copy. Please see the lesson on copy constructors for an explanation of shallow and deep copies.

Operator Overloading

This lesson covers the topic of operator overloading. Operator overloading provides a way to define and use operators such as +, -, *, /, and [] for user defined types such as classes and enumerations. By defining operators, the use of a class can be made as simple and intuitive as the use of intrinsic data types.

Let's develop a "fraction" class to illustrate the use and utility of operator overloading. First, consider which operators are normally associated with fractions and should be included in this class. Operators such as +, -, *, / are givens. C++ allows any of its built in operators to be overloaded in a class, but only those that have an intuitive meaning for a particular class should actually be overloaded. For instance, although the modulo operator, %, could be overloaded for our fraction class, its meaning would be unclear and defining it would only confuse users of the class. %1/2 has no clear interpretation.

To begin, here is an implementation of the fraction class with an add method. Note that in order to simplify the implementation of the add method, I am using the greatest common denominator of the two fractions rather than the least.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
```

```
    Fraction add(const Fraction &rhs)
    {
        Fraction temp;
        temp.den = this->den * rhs.den;
        temp.num = rhs.den * this->num +
            this->den * rhs.num;
        return temp;
    }
    void print()
    {
        cout << num << "/" << den << endl;
    }
private:
    int num;
    int den;
};

int main() {
    Fraction a(1,2);
    Fraction b(1,4);
    Fraction c;

    a.print();
    b.print();

    c = a.add(b);

    c.print();

    return 0;
}
```

Output:
1/2
1/4
6/8

Notice that a temporary Fraction object was created within the add method. This was necessary because "add" must return a Fraction, since its result is assigned to a Fraction. The use of the add method is somewhat awkward. Rather than a.add(b), we'd like to be able to write "a + b". This can be done with operator overloading, as shown on the following page.

Here is the Fraction class with an overloaded "+" operator, rather than an "add" method as on the previous page.

```
#include <iostream>
using std::cout;
```

```
using st::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction operator+(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num +
         this->den * rhs.num;
      return temp;
   }
   void print()
   {
      cout << num << "/" << den << endl;
   }
private:
   int num;
   int den;
};

int main() {
   Fraction a(1,2);
   Fraction b(1,4);
   Fraction c;

   a.print();
   b.print();

   c = a + b;

   c.print();

   return 0;
}
```

Output:
1/2
1/4
6/8

The syntax for overloading an operator is at first a bit intimidating. But if you look closely, you'll see that the only change made in the fraction class is that the method

name "add" was replaced by the keyword operator followed by a "+".

Here is the syntax for overloaded operators, where op is the operator being overloaded:

```
return_type operator op (parameter list);
```

Practice: Add overloaded operators for "-", "*" and "/" to the Fraction class.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction operator+(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num +
         this->den * rhs.num;
      return temp;
   }
   Fraction operator-(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num -
         this->den * rhs.num;
      return temp;
   }
   Fraction operator*(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = this->num * rhs.num;
      return temp;
   }
   Fraction operator/(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.num;
```

```
        temp.num = this->num * rhs.den;
        return temp;
    }
    void print()
    {
        cout << num << "/" << den << endl << endl;
    }
private:
    int num;
    int den;
};

int main() {
    Fraction a(1,2);
    Fraction b(1,4);
    Fraction c;


    cout << "a is ";
    a.print();
    cout << "b is ";
    b.print();

    cout << "a + b = ";
    c = a + b;
    c.print();

    cout << "a - b = ";
    c = a - b;
    c.print();

    cout << "a * b = ";
    c = a * b;
    c.print();

    cout << "a / b = ";
    c = a / b;
    c.print();

    return 0;
}
```

Results:
a is 1/2

b is 1/4

a + b = 6/8

a - b = 2/8

a * b = 1/8

a / b = 4/2

Since we have defined +, -, *, / for the Fraction class, we should also define the compound operators, +=, -=, *= and /=. Unfortunately, they must be explicitly defined; the compiler does not create them from the overloaded methods defined above plus the default copy assignment. Here is an implementation for "+=".

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction& operator+=(const Fraction &rhs)
   {
      this->num = rhs.den * this->num +
         this->den * rhs.num;
      this->den = this->den * rhs.den;
      return *this;
   }
   void print()
   {
      cout << num << "/" << den << endl;
   }
private:
   int num;
   int den;
};

int main() {
   Fraction a(1,2);
   Fraction c(1,1);

   cout << "a is ";
   a.print();
   cout << endl;

   cout << "c is ";
```

```
    c.print();
    cout << endl;

    c += a;
    cout << "c += a is ";
    c.print();

    return 0;
}
```

Practice: Define overloaded operators for -=, *=, /=.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
    Fraction(int num = 0, int den = 1)
    {
        this->num = num;
        this->den = den;
    }
    Fraction& operator+=(const Fraction &rhs)
    {
        this->num = rhs.den * this->num +
            this->den * rhs.num;
        this->den = this->den * rhs.den;
        return *this;
    }
    Fraction& operator-=(const Fraction &rhs)
    {
        this->num = rhs.den * this->num -
            this->den * rhs.num;
        this->den = this->den * rhs.den;
        return *this;
    }
    Fraction& operator*=(const Fraction &rhs)
    {
        this->num = this->num * rhs.num;
        this->den = this->den * rhs.den;
        return *this;
    }
    Fraction& operator/=(const Fraction &rhs)
    {
        this->num = this->num * rhs.den;
        this->den = this->den * rhs.num;
```

```
    return *this;
    }
    void print()
    {
        cout << num << "/" << den << endl;
    }
private:
    int num;
    int den;
};
```

Next, let's examine how to implement a prefix increment operator, "++", for the Fraction class. For an object of class Fraction, a, we will define ++a. A sensible definition of this operation is to increment the fraction by one. As a first try, let's write a prefix increment similarly to the way the overloaded "+" was written.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
    Fraction(int num = 0, int den = 1)
    {
        this->num = num;
        this->den = den;
    }
    void operator++()
    {
        num += den;
        // For example ++(1/2) = (1 + 2)/2 = 3/2
    }
    void print()
    {
        cout << num << "/" << den << endl;
    }
private:
    int num;
    int den;
};

int main() {
    Fraction a(1,2);

    cout << "a is ";
    a.print();
```

```
    cout << "++a is ";
    ++a;
    a.print();

    return 0;
}
```

Results:
a is 1/2
++a is 3/2

This implementation of Fraction works for the sample program, but there is are problems, as described on the next page.

Suppose we want to assign the results of the increment to another Fraction.

```
c = ++a;
```

This will give an error during compilation since the overloaded increment operator is returning void. We need it to return an object of type fraction. To fix this, let's try using a temp Fraction object as was done earlier in this lesson.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
    Fraction(int num = 0, int den = 1)
    {
        this->num = num;
        this->den = den;
    }
    Fraction operator++()
    {
        num += den; //First increment the calling object.
        // For example ++(1/2) = (1 + 2)/2 = 3/2
        // Create a temp object
        Fraction temp(num, den);
        // Return the temp object
        return temp;
    }
    void print()
    {
        cout << num << "/" << den << endl;
    }
```

```
private:
   int num;
   int den;
};

int main() {
   Fraction a(1,2);
   Fraction c;

   cout << "a is ";
   a.print();

   cout << "c is ";
   c = ++a;
   c.print();

   return 0;
}
```

Results:
a is 1/2
c is 3/2

Since the temporary object is never used within the prefix increment method, there is no need to create a named object. To reduce clutter in the code, a better technique is to return the results of the Fraction constructor directly.

```
Fraction operator++()
{
   num += den;
   return Fraction(num, den);
}
```

Let's consider the returned value more carefully. Is the temporary object really required at all? Every method of a class is passed a pointer to the calling object, the "this" pointer. Rather than create a temporary object, the calling object can be returned by dereferencing the "this" pointer. Since "this" is a pointer, *this is used to dereference and return the object itself.

```
Fraction& operator++()
{
   num += den;
   return *this;
}
```

Notice that the return type of the method had also been modified. This is not strictly

necessary but does avoid the cost of an additional copy. With the reference, the assignment is direct. If the method was declared to return Fraction rather than Fraction& an additional copy is made of the returned object. This copy is the assigned to the left hand side of the expression.

Let's continue with this example by defining a postfix increment operator. Huh? Above, we defined a prefix increment operator.

```
c = ++a;    //Prefix increment
```

This says increment a, then assign the result to c.

```
c = a++;    //Postfix increment
```

This says assign the current value of a to c, then increment a.

There is a problem. The operator, in both cases, is "++". We cannot just create a two overloaded operator methods, both with the same declaration.

```
Fraction& operator++();
```

The signature of each method within a class must be unique; otherwise the compiler barks. To create a unique signature, a "dummy" int parameter is added to the parameter list of the postfix operator method. This is solely to each method to have a unique signature and is arbitrary. There is no real rhyme or reason. You must simply need to remember that the postfix implementation has this additional parameter.

```
Prefix: Fraction& operator++();
Postfix: Fraction operator++(int);
```

Here's the implementation.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction& operator++()
```

```
    {
        num += den;
        return *this;
    }
    Fraction operator++(int)
    {
        Fraction temp(num, den);
        num += den;
        return temp;
    }
    void print()
    {
        cout << num << "/" << den << endl;
    }
private:
    int num;
    int den;
};

int main() {
    Fraction a(1,2);
    Fraction c;

    cout << "a is ";
    a.print();
    cout << endl;

    cout << "Prefix Increment" << endl;
    c = ++a;
    cout << "c is ";
    c.print();
    cout << "a is ";
    a.print();
    cout << endl;

    cout << "Postfix Increment" << endl;
    c = a++;
    cout << "c is ";
    c.print();
    cout << "a is ";
    a.print();

    return 0;
}
```

Results:

a is 1/2

Prefix Increment
c is 3/2
a is 3/2

Postfix Increment
c is 3/2
a is 5/2

Notice that the postfix increment method requires a temp object to store the value prior to incrementation. Also notice that this method returns its result by value while the prefix method returns its result by reference. Why? For the postfix method, the temporary object goes out of scope (ceases to exist) when the method returns. Returning a reference to an out of scope object is a big bug. You may get the correct results, incorrect results or even program death. By returning by value, a copy is made and this issue is avoided. For the prefix method, the calling object exists before, during and after the method execution, so a reference can be used to avoid extra copies.

Practice: Try implementing postfix and prefix decrement operators for the Fraction class.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction& operator--()
   {
      num -= den;
      return *this;
   }
   Fraction operator--(int)
   {
      Fraction temp(num, den);
      num -= den;
      return temp;
   }
   void print()
   {
      cout << num << "/" << den << endl;
   }
private:
```

```
    int num;
    int den;
};

int main() {
    Fraction a(1,2);
    Fraction c;

    cout << "a is ";
    a.print();
    cout << endl;

    cout << "Prefix Increment" << endl;
    c = --a;
    cout << "c is ";
    c.print();
    cout << "a is ";
    a.print();
    cout << endl;

    cout << "Postfix Increment" << endl;
    c = a--;
    cout << "c is ";
    c.print();
    cout << "a is ";
    a.print();

    return 0;
}
```

This is the second part of a lesson on operator overloading. It presents some general rules, operators in global and namespace scope and the concept of friend functions. Overloaded operators can be declared with within a class or outside in global or namespace scope. In C++ a function declared as a friend to a class can access and manipulate the non-public members of that class. Classes can also be declared as friends of a class as will be described in a later lesson.

General Rules for Operator Overloading

1. Only existing operator symbols may be overloaded. New symbols, such as ** for exponentiation, cannot be defined.
2. The operators ::, .*, . and ?: cannot be overloaded.
3. Operators =, [], () and -> can only be defined as members of a class and not as global functions.
4. At least one operand for any overload must be a class or enumeration type. It is not possible to overload operators involving only built-in data types. For example, an attempt to overload addition, +, for the int data type would result in a compiler error. int operator+(int i, int j) is not allowed.

5. The arity or number of operands for an operator may not be changed. For example, addition, +, may not be defined to take other than two arguments regardless of data type.
6. The precedence of operators is not changed be overloading.

Overloaded Operators in Global and Namespace Scope

In addition to being defined in class scope (within a class), overloaded operators may be defined in global or namespace scope. Global scope means that the operator is defined outside of any function (including main) or class. Namespace scope means that the operator is defined outside of any class but within a namespace, possible within the main program.

Let's return to the Fraction class introduced in the last lesson. This time, we will enhance the class to allow mathematical operations between Fractions and integers. As a start, on the next page is the Fraction class with an added overloaded operator to handle the addition of an integer ***to*** a Fraction.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction operator+(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num +
         this->den * rhs.num;
      return temp;
   }
   Fraction operator+(const int i)
   {
      Fraction temp;
      temp.den = this->den;
      temp.num = this->num +
         this->den * i;
      return temp;
   }
   void print()
   {
      cout << num << "/" << den << endl;
   }
private:
   int num;
```

```
    int den;
};

int main() {
    Fraction a(1,2);    // a = 1/2
    int b = 5;
    Fraction c;

    c = a + b;
    c.print();

    return 0;
}
```

Results:
11/2

Notice that the overloaded operator+ has been overloaded itself to accept either a Fraction or int argument. Overloaded operators are just methods; they can be overloaded if the number and/or type of their arguments differ for each version.

So, we have defined an operator that will add an int ***to*** a Fraction. Will this also allow a Fraction to be added ***to*** an int? No. To understand why let's put on special glasses that allow us to see things as the compiler does.

In the main program, above, a is a Fraction and b is an integer. The line

```
c = a + b;
```

is expanded by the compiler as

```
c = a.operator+(b);
```

That is, the overloaded operator for + is called on the Fraction object "a" with an argument equal to "b".

Let's see how the compiler expands c = b + a;

```
c = b + a;
c = b.operator+(a);
```

It attempts to call the operator "+" on an int object with a Fraction as its argument. No such operator exists for ints. We will need to develop an operator outside the Fraction class, in namespace or global scope, which takes an int and a Fraction as arguments. This is shown on the next page.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction operator+(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num +
         this->den * rhs.num;
      return temp;
   }
   Fraction operator+(const int i)
   {
      Fraction temp;
      temp.den = this->den;
      temp.num = this->num +
         this->den * i;
      return temp;
   }
   int getDen() const {return den;}
   int getNum() const {return num;}
   void setDen(int d) {den = d;}
   void setNum(int n) {num = n;}
   void print()
   {
      cout << num << "/" << den << endl;
   }
private:
   int num;
   int den;
};

Fraction operator+(const int i, const Fraction f)
{
   Fraction temp;
   int denominator;
   int numerator;
   denominator = f.getDen();
   numerator = f.getNum() +
   f.getDen() * i;
```

```
    temp.setDen(denominator);
    temp.setNum(numerator);
    return temp;
}
int main() {
    Fraction a(1,2); // a = 1/2
    int b = 5;
    Fraction c;

    c = a + b;
    c.print();

    c = b + a;
    c.print();

    return 0;
}
```

Results:
11/2
11/2

Notice that just before main an additional overloaded operator has been defined in global scope. The compiler correctly resolves a + b, where a is an int and b is a Fraction, to this method. Also, notice that since this new overloaded operator is not part of the Fraction class it cannot access the non-public data members of that class. Nor would it be able to access any non-public methods, if any existed. Accessor methods to get and set the private data members were added to the Fraction class to allow the global overloaded operator to access the private members of that class. In the next section, we'll see how defining this operator as a "friend" to the class Fraction can simplify coding.

Friends
A method declared as a friend to a class can access the non-public members and methods of that class, as well as the public members and methods of that class. This can simplify coding, but at a cost. A basic tenant of object-oriented programming is data hiding or encapsulation. Functions and objects outside of a class do not need and probably should not have knowledge of how a class stores its data. A change in the way data is stored within the class is isolated from the users (other classes and functions) of the class. Declaring a method as a friend removes this isolation. Generally, avoid friends. As we'll see in a later lesson, classes can also be declared friends of each other.

The argument for declaring methods as friends of classes is that sometimes a method is intimately related to a class but cannot be declared as a member of that class. This does describe the case in the previous example. The overloaded method for "+" cannot be part of the Fraction class since it is not called on a Fraction object and it is certainly very closely related to the Fraction class. Whether to declare it as a friend or to use accessor methods is a judgment call.

A method is defined to be a friend of a class by declaring it with the keyword "friend" within the definition of the class. Here is our example with the overloaded operator declared as a friend. Notice that the accessor methods are no longer needed and have been removed.

```
#include <iostream>
using std::cout;
using std::endl;

class Fraction {
public:
   Fraction(int num = 0, int den = 1)
   {
      this->num = num;
      this->den = den;
   }
   Fraction operator+(const Fraction &rhs)
   {
      Fraction temp;
      temp.den = this->den * rhs.den;
      temp.num = rhs.den * this->num +
         this->den * rhs.num;
   return temp;
   }
   Fraction operator+(const int i)
   {
      Fraction temp;
      temp.den = this->den;
      temp.num = this->num +
      this->den * i;
      return temp;
   }
   void print()
   {
      cout << num << "/" << den << endl;
   }

   friend Fraction operator+(const int i, const Fraction f);

private:
   int num;
   int den;
};

Fraction operator+(const int i, const Fraction f)
{
   Fraction temp;
```

```
    temp.den = f.den;
    temp.num = f.num() +
        f.den() * i;
    return temp;
}

int main() {
    .....
}
```

Relationships Between Classes In C++

You have learned the language, studied its syntax, perhaps written small programs as assignments from classes, web tutorials or books and are now ready to tackle a bigger project. What is the first step? Before any programs are written, really before a single character is typed in a file, some design work must be done. For an object-oriented language, such as C++, design starts with choosing classes and defining their relationships.

The three main types of relationships between classes are generalization (inheritance), aggregation, and association.

- **Generalization** - This implies an "is a" relationship. One class is derived from another, the base class. Generalization is implemented as inheritance in C++. The derived class has more specialization. It may either override the methods of the base, or add new methods. Examples are a poodle class derived from a dog class, or a paperback class derived from a book class.
- **Aggregation** - This implies a "has a" relationship. One class is constructed from other classes, that is, it contains objects of any component classes. For example, a car class would contain objects such as tires, doors, engine, and seats.
- **Association** - Two or more classes interact in some manner. They may extract information from each other, or update each other in some way. As an example, a car class may need to interact with a road class, or if you live near any metropolitan area, the car class may need to pay a toll collector class.

Assume that you've been asked by the local zoo to write a program that will be used to study animals in order to create better habitats. What classes might be needed? An animal class is a good starting point.

```
class Animal {
private:
    int itsAge;
    float itsWeight;
public:
    // Accessor methods have been left out as a simplification
    void move() {cout << "Animal Moving\n";}
    void speak() {cout << "Animal Speaking\n";}
```

```
    void eat() {cout << "Animal Eating\n";}
}
```

To better study a particular type of animal, it is necessary to **generalize**.

```
class Duck: public Animal {
private:
public:
    // Accessor methods have been left out as a simplification
    void move() {cout << "Waddle\n";}
    void speak() {cout << "Quack\n";}
}
```

The Duck class has inherited some methods and members from Animal, and has made some of its methods more specialized. Not everyone can waddle and quack. An animal consists of certain parts: for instance, head, body, and skin. It is an **aggregation** of these parts. The animal class, likewise, can be an aggregation of many other classes.

```
class Animal {
private:
    int itsAge;
    float itsWeight;
    Head itsHead;
    Body itsBod;
    Heart itsHeart;
public:
    // Accessor methods have been left out as a simplification
    void move() {cout << "Animal Moving\n";}
    void speak() {cout << "Animal Speaking\n";}
    void eat() {cout << "Animal Eating\n";}
}
```

Association is a logical relationship. The classes need to know each others interfaces and need to interact but there is no formal "is a" or "has a" relationship as in generalization and aggregation, respectively. As an example, a zookeeper class would need associations with the animal classes. Interactions between these classes would be seen through out the zoo program.