LECTURE 3

Classes

C++ is an object-oriented language. It's based on classes. To define a class is to define a new type (like an `int` or a `double`) which you can then use just like it was built into the language. A class should embody some *conceptual* or *mathematical object*. Choosing which classes to make is the most important part of designing a C++ program. It is an art that can be learned.

When you make a class, you bury the details inside the class that you don't want to think about everytime you use it. You use the class in a high level way.

Let me give an analogy. (This is from *C++ for Dummies* by Stephen Davis.) Suppose you want to make nachos one night. So you dump some chips on a plate, throw on some beans, cheese, and jalapenos, and nuke the mess in the microwave oven for a few minutes. To use the microwave, you open the door, throw the stuff in, and punch a few buttons on the front. After a few minutes, the nachos are done. Notice that you don't look inside the case of the microwave or try to rewire it. You don't have to reprogram the software to use the microwave everytime you cook a different dish. The microwave has an interface consisting of a front panel with all the buttons you need to push. You work at a certain level of detail where you don't have to worry about the inner workings of the microwave. The level of detail at which you are working is called the *level of abstraction*. We can say that the microwave oven allows us to *abstract away* the details of the microwave's internals. The microwave oven is like a class in C++.

Let's compare and contrast the functional programming approach (what you do in C and Fortran) with the object–oriented approach (C++). In a functional programming approach to making nachos, a flow chart of the program would show the program going from your finger to the front panel and then to the internals of the microwave. Pretty soon, flow would be wiggling around through complex logic paths about how long to turn on the Klystron and whether to sound the "come and get it" tone.

In an object–oriented approach to making nachos, you would first identify the types of objects in the problem: chips, beans, cheese and an oven. These are the analogs of basic classes. Then you would begin the task of modeling these objects in software, without regard to how they will be used in the final program. If you work at the level of basic objects, you think about making a useful oven, and not about making some specific snack like nachos. At the next level of abstraction you make nachos.

In object–oriented computerese, your microwave is an *instance* of the class microwave. The class microwave is a subclass of the class oven, which in turn is a subclass of the class kitchen appliances.

Classes can get extremely complicated, but let's start with simple examples of mathematical objects: "a point in a plane" and "a line in a plane". (see Chapter 4 of Barton and Nackman)

Definition of class Point

```
//          This is the file Point.h

  typedef double Real; //This means that Real is a synonym for double.

  class Line; //Declaration of class Line, which has not
                              // yet been defined. But we need to tell
                              // class Point that it exists because
                              // class Point uses it.
  class Point{
  public:                     // "public" means accessible to the outside
                              // world, i.e., the parts
                              // of the program outside this class.

     Point();                 // Declare a function Point() with no
                              // arguments. This is actually a special
                              // function called a class constructor
                              // that creates an object of the class Point.

     Point(Real x, Real y);   // Constructor that creates an object using
                              // the values x and y.

     Real distance(Point p);  // Member function that computes the distance
                              // to another point from this point. It
                              // returns a Real.

     Real distance(Line L);   // Member function that computes the distance
                              // to a line.

     Real x();                // Get x-coordinate of point.

     Real y();                // Get y-coordinate of point.

   private:                   // Accessible only to class objects.

    Real xc;                  // x-coordinate.
    Real yc;                  // y-coordinate.
   };                         // Only place a ";" follows a bracket "}"
```

Use of class Point

```
  #include <iostream.h>
  #include "point.h"
```

```
void main()
  {
  Point origin(0.,0.);       //call constructor of Point
  Real x, y;
  cin >> x >> y;
  Point p(x,y);              //call constructor of Point

  cout << origin.distance(p) << endl;   //distance to point p from origin
  }
```

An "object" is a specific instance or realization of a class. `origin` and `p` are objects of the class Point. The period "." in `origin.distance(p)` is a class member access operator. This is `origin`'s distance function. It is similar in form to `origin.distance(Line L)`. C++ decides on the basis of the arguments type which function to call. This is called *function overloading*.

Private versus Public

Consider the following:

```
main()
 {
 Point p;
 cout << p.xc << endl;       //Error: can't access private data members
 }
```

This would be ok if `xc` and `yc` were declared in the `public` sector. Why use `private`? So no one can corrupt your data members. Private members represent the internal workings of the class. Public members, on the other hand, represent the interface of the class with the rest of the world. It's a good idea to hide the details of the class in private so that when you want to change those details, it's less likely to require changes in the way the class is used in external applications. An example of internal details would be the way the data is stored. For example the coordinates of a point could be stored in Cartesian coordinates or in polar coodinates. Private is the default. So we could define the class Point as follows:

```
class Point{
    Real xc;                 //Private data member
    Real yc;                 //Private data member

  Public:
    Point();
     ...

};
```

A point with 2 data members (`double xc, double yc`) is stored as $2 * 8 = 16$ bytes. Functions take no memory other than in the executable code.

class Line

```
class Line
  {
public:
  Line(Point p1, Point p2);           //Line through 2 points
  Line(Point p, Real xdir, Real ydir);
                                  // Line through p with tangent (x,y)
  Point intersect(Line line);
  Real distance(Point p);
  static Real parallelism_tolerance;
private:
  Real a;                         // ax + by + c = 0
  Real b;
  Real c;
  };
```

Use:

```
main()
 {
 Line L1(Point(0,0), Point(0,1));
 Line L2(Point(1,2), 1,1);
 point intersection  = L1.intersect(L2);
 cout << "(" << intersection.x() << "," << intersection.y() << ")"
      << endl;
```

Definition of Member Functions

```
// This is the file point.cc

 Point::Point()                   // Scope resolution operator ::
  {}                              //constructor does nothing

 Point::Point(Real x, Real y)
  {
  xc=x;                           // Member function
                                  // has access to private members
  yc=y;
  }
```

The :: is called the *scope resolution operator* because it indicates to which class a member belongs. You also can use the :: operator with a non–member function as well as using a null structure name. For example, if `power(float x, float y)` is a not a member of any class and is declared globally, we could write `::power(float x, float y)`. This is optional except when 2 functions of the same name exist. Then you need to specify if you are calling the global function.

Class Constructors

A *constructor* is a member function that has the same name as the class. `Point::Point()` and `Point::Point(Real x, Real y)` are constructors. The constructor is called automatically when an object of its class is created. Its primary job is to initialize the object to a legal initial value for the class. If an array is created, e.g. `Point p[5]`, the default constructor is called 5 times. The constructor has no return type, not even *void*. If you don't provide a constructor for a class, the C++ will automatically provide one for you. It's called the *default*, or *void*, constructor. The default constructor sets all the data members of the object to binary zero. If your class already has a constructor, C++ doesn't provide the automatic default constructor. So if you define a constructor for your class, but you also want a default constructor, you must define it yourself. `Point::Point()` is a default constructor. The default constructor is called when an array of objects is created, so it's a good idea to have a default constructor. Note that the compiler–supplied default constructor won't work if you have `const` or reference variables that need to be initialized in the class.

Unlike other functions, you can't call a constructor; it's called automatically when the object is created. Therefore, the only way to pass arguments to the constructor is when the object is created. So the statement `Point origin(0.,0.)` in main calls the constructor `Point::Point(Real x, Real y)` and passes the values $(0., 0.)$ to (x, y). `Point p(3.0,2.0)` creates 16 bytes of space and calls this function. The declaration `Point p` gets 16 bytes and calls the function `Point()`. Notice that in `main`, you write `Point p`, not `Point p()`, in order to invoke the default constructor. If you write `Point p()` in `main`, you are declaring a function that returns an object of class `Point` by value. Note that a constructor should be a public member function because it will be called by outside function like `main`.

Constructing a Data Member

Suppose you have a data member that itself is an object of a class. For example, suppose you have

```
class Map
 {
 private:
   Point irvine;
 }
```

How does one initialize the `Point irvine`? In other words, how does one pass arguments to the `Point` constructor?

The data members are constructed *before* the body of the `Map` constructor is entered. So we can't write

```
class Map
 {
 public:
   Map(real x, real y)
     {Point irvine(x,y);}          //ERROR-don't do this.
 private:
   Point irvine;
 }
```

This would create the Point irvine twice; once before the body of the Map constructor is entered and once after the body of the Map constructor is entered. The second one disappears when the program returns from the Map constructor. We only want one irvine constructed, and we want it to be the data member of Map.

The correct way to initialize a data member is as follows:

```
class Map
 {
 public:
   Map(real x, real y):irvine(x,y)
   {}
 private:
   Point irvine;
 }
```

The ":" means that what follows are calls to the constructors of data members of the current class. To the C++ compiler, this line reads, "Construct the member `irvine` using the arguments `x` and `y` of the `Point` constructor." We invoke this constructor as follows:

```
main()
 {
 Map county_map(38.5,24.2);
 }
```

If no values are given when a Map object is created, we can invoke default values by writing the constructor as follows:

```
class Map
 {
Map(real x=12.5, real y=18.0):irvine(x,y)
   {}
 private:
   Point irvine;
 }
```

where (x=12.5, y=18.0) are the default values. If no arguments are given, this constructor acts as a default constructor.

The ":" syntax must also be used to assign values to const and reference type members. For example:

```
class SillyClass
 {
 public:
   SillyClass(int& i):ten(10),refI(i)
   {}
 private:
   const int ten;
   int& refI;
 };

 main()
 {
  int i;
  SillyClass sc(i);
  return 0;
 }
```

Point.cc Continued

```
Real Point::x()
 {return xc;}
Real Point::y()
 {return yc;}
```

The distance to another point is simple too.

```
Real Point::distance(Point p)
  {
  Real xdif = xc - p.xc;    // xc refers to the xc of "this" object
  Real ydif = yc - p.yc;    // p.yc refers to the yc of the other object p.
                            //Notice access of the private data member
                            //p.yc by another object of the same class.

  return sqrt(xdif*xdif + ydif*ydif);
  }

Real Point::distance(Line L)
  {
  return L.distance(*this);   // Let Line do the work.
```

```
  }

Real Line::distance(Point point)
  {
  //Returns the distance from point to the line.
  return abs(a*point.x() + b*point.y() + c)/sqrt(a*a + b*b);
  }
```

`a, b` and `c` are intialized in the constructors of `Line` which can be found on page 93 of the textbook.

`this` is a pointer to the "current" object. For example, in `p1.distance(L)`, `this` is a pointer to `&p1`. You don't have to write

```
Point * const this = &p1;
```

because the compiler already has done it for you. Above, in the expression `xdif = xc - p.xc`, we could have written `xdif = (*this).xc - p.xc`. The `(*this).xc` notation is awkward, so there is another notation that means the same thing:

```
this -> xc
```

In general, use `->` to access a data member or a member function associated with a pointer to a class. For example, if a class (say Line) has a member function `f(s)` or a data member `a`, then

```
Real v;
Line *LL;          // LL is a pointer to a Line object.
v = LL -> a;       // LL->a accesses the data member "a" of the
                   // class object pointed to by LL. This is the
                   // same as (*LL).a

LL -> f(s);        // Calls the function f(s) which is a member of
                   // object pointed to by LL. This is the same as
                   // (*LL).f(s).
```

Static Members of a Class

A static variable in a class is shared by all the objects created with that class type. For example, if you have a class `student`, the number of students in the course would be a static variable that is the same for all objects of the class. The number of students is the number of objects of the class. The access rules for static members are the same as the access rules for normal members. From within the class, static members are referenced like any other class member. Public static members can be referenced from outside the class as well. A static data member is associated with a class and not with any particular object of that class. So a reference to a static member function does not require an object. If an object is present, only its type is used. Consider the following example:

```
class Student{
 public:
   Student()                        //Default constructor
     {
     noOfStudents++;                //reference from inside the class
     }
   static int noOfStudents;
      //other stuff
 };

 int Student::noOfStudents=0;    //Define and initialize static variables
                                 //outside the class declaration.
                                 //Notice the word "static" is absent.

 void fn(Student &s1, Student &s2)
  {
     //The following references from outside the class produce identical
     //results.

     cout << "Number of Students" << s1.noOfStudents << endl;
     cout << "Number of Students" << s2.noOfStudents << endl;
     cout << "Number of Students" << Student::noOfStudents << endl;
  }
```

Note that `static` means something different from `const`. Static variables can change as the program progresses. You can use static members to keep count of the number of objects floating around. A static variable can also be used as a flag to indicate whether a particular action has occurred. Finally, a very common use for static members is to contain the pointer to the first member of a linked list.

Static variables must be defined outside the declaration of the class. So if the class is defined in a `.h` file, you should put the static variable definition in the .cc file.

```
// This is the file line.cc.

Real Line::parallism_tolerance=0.1;
```

Notice that the word `static` does not appear in the definition. `parallism_tolerance` is used in the definition of `intersect()`:

```
Point Line::intersect(Line line) {
    // Returns the point where this line intersects with another line.
    // If the angle between the two lines is less than
    // the parallelism_tolerance, return the point at infinity. The parallelism
    // test computes the square of the sin of the angle. We assume that the
```

```
    // tolerance is small enough for sin(theta) to be
    // approximately equal to theta.

    Real det = a * line.b - line.a * b;
    Real sinsq = (det * det)/((a*a + b*b) * (line.a*line.a + line.b*line.b));
    if (sinsq < parallelism_tolerance * parallelism_tolerance) {
        return Point(FLT_MAX, FLT_MAX);
                    // Point at infinity (FLT_MAX from float.h)
    }
    else {
        return Point((b * line.c - line.b * c)/det,
                    (c * line.a - line.c * a)/det));
    }
}
```

Static Member Functions

There are also static member functions, which are associated with a class and not with any particular object of that class. This means that like a reference to a static data member, a reference to a static member function does not require an object. If an object is present, only its type is used. Consider the following example:

```
#include <iostream.h>
#include <string.h>
class Student
{
 public:
   Student()                      //Default constructor
     {
     noOfStudents++;              //reference from inside the class
     }
   static int number()
     {
       return noOfStudents;
     }
 private:
   static int noOfStudents;
   char name[40];
      //other stuff
 };
 int Student::noOfStudents=0;     //Initialize static data member
 int main()
 {
   Student s;
```

```
  // The following calls produce identical results.
  cout << s.number() << endl;
  cout << Student::number() << endl;
  return 0;
}
```

Notice how the static member function can access the static data member. A static member function is not directly associated with any object, however, so it does not have default access to any non–static members. Thus, the following would not be legal:

```
class Student
{
  public:
  //The follwing is not legal
  static char *sName()
  {return name;}        //which name? There is no object.
  //other stuff ...

  private:
  char name[40];
  static int noOfStudents;
 };
```

That is not to say that static member functions have no access to non–static data members. For example, a static member function could be used to search through a linked list of objects of the class. Consider the following:

```
#include <iostream.h>
#include <string.h>
class Student
{
 public:
   Student(char *pName = "no name")      //constructor
                                         //("no name" is default value)
     {
     strcpy(name, pName);
     noOfStudents++;
     }
   ~Student()                            //destructor (see Lecture 4)
     {
       noOfStudents--;
     }

    //findName - return student with specified name
```

```
     static Student *findname(char *pName);
private:
  static int noOfStudents;
  static Student *pFirst;
  Student *pNext;
  char name[40];
};
int Student::noOfStudents=0;     //Initialize static data member
Student* Student::pFirst=0;      //Initialize static data member

//findName- return student with specified name.
//          Return zero if no match.
Student* Student::findName(char *pName)
{
  //loop through linked list...
  for(Student *pS=pFirst; pS; pS = pS->pNext)
  {
    //if we find the specified name...
    if(strcmp(pS->name, pName) == 0)
      {
        //then return the object's address
        return pS;
      }
  }
  //otherwise, return a zero (item not found)
  return (Student*)0;              //cast
}

int main()
{
  Student s1("Randy");
  Student s2("Jenny");
  Student s3("Susy");
  Student *pS = Student::findName("Jenny");
  return 0;
}
```

The function `findName()` has access to `pFirst` because it's shared by all objects. Being a member of class`Student`, `findName()` has access also to `name`, but the call must specify the object to use (that is, whose name). No default object is associated with a static member function. Calling the static member function with an object doesn't help. For example:

```
//...same as before...
    int main()
  {
    Student s1("Randy");
    Student s2("Jenny");
    Student s3("Susy");
    Student *pS = s1.findName("Jenny");
    return 0;
  }
```

The `s1` is not evaluated and not passed to `findName()`. Only its class is used to decide which `findName()` to call.