Classes and Objects

Object-oriented programming (OOP) is the next step in programming practice after functional programming (1960's), modular programming (1970's), and later implementations of data encapsulation and software reusability concepts (1980's).

The first main principle of OOP is that of the class. Programming with classes is the basis of OOP.

Classes

A class is a kind of package that consists of functions plus declarations of the data variables they use. Functions are called *methods* in OOP, and individual variables declared in a class are called *data members*. A class may be viewed as a new data type.

Objects

A class acts as a template for creation of objects of a new type. Each object instance defines a set of the data members declared within the class for itself. Objects can then use common class methods.

From Struct to Class

For example, a type for holding data of birth records could be made by combining string type variables for names with int type variables for birth date. These combined variables are called *structures* in C.

The OOP approach adds the concept of class, which includes functions as well. A member function to calculate the age can be included into the birth record type. When a particular variable of that type (object) has its age member function called, the function will automatically use the data members of that particular object to calculate the age.

Programming with Classes

To illustrate these concepts, a C++ program example that does not use object-oriented techniques is considered first.

Non OOP Version:
C++ program using a function to calculate area of rectangle

```
int rectArea(int xa, int ya, int xb, int yb) {
  // xb>xa and yb>ya
  return (xb-xa)*(yb-ya);
}

int main() {
  int x1, y1, x2, y2;
  x1 = 12; y1 = 23; x2 = 34; y2 = 45;
  cout << rectArea(x1, y1, x2, y2);
  return 0;
}
```

Note that to calculate the area of one rectangle, one needed:

- **4 variables to define the rectangle**
- **4 initialization statements**
- **1 function call with 4 arguments named in the right order**

To calculate the area of each additional rectangle, one needs:

- **4 more variables**
- **4 more initialization statements**
- **1 more function call**

Also, one needs a way to keep track of all the coordinates by some naming convention like R1x1 for x1 of rectangle 1, etc. If a well planned naming system is not adopted, it can get very confusing with many variable names.

To simplify this, a structure may be used for the x and y variables, and the rectArea function may be improved by using one argument only:

Non-OOP Version with struct:
C++ program using a Rectangle struct

```
struct Rectangle {
  int xa, ya, xb, yb;   // xb>xa and yb>ya
};

int rectArea(Rectangle rec) {
  return (rec.xb-rec.xa)*(rec.yb-rec.ya);
}

int main() {
  Rectangle R;
  R.xa = 12; R.ya = 23; R.xb = 34; R.yb = 45;
  cout << rectArea( R );
  return 0;
}
```

So in this improved version, one needed:

- 1 variable of type Rectangle (containing 4 members)
- 4 initialization statements
- 1 simple function call with one argument only

For each additional rectangle, one needs:

- 1 additional variable of type Rectangle (with same members)
- 4 additional initialization statements
- 1 additional simple function call

In referencing a rectangle in calls, the names of its four data members are not needed. The single argument needed for a function call is the name of the Rectangle only.

For an OOP Version of the program, a class called Rectangle is defined. The class includes not only the variables that represent the rectangle coordinates, but also the code that calculates its area. The class definition is a little like a struct plus some function prototypes.

C++ Rectangle Class Definition

```
class Rectangle {
  private:
    int x1, y1, x2, y2;                 // the data members

  public:
    Rectangle(int, int, int, int);      // the "constructor"
    int area();                         // the area method
};                                      // note ;
```

The Rectangle class thus contains 4 data members and 2 methods. The constructor is called whenever a Rectangle instance (object) is created. Its arguments are copied into the data members of the object for initialization. The constructor name is the same as the class name.

Code for the class constructor and the area methods can be written outside the Rectangle class definition.

C++ Code for Rectangle Class Methods

```
Rectangle::Rectangle(int left, int bot, int right, int top)
{
  x1 = left; y1 = bot;
  x2 = right; y2 = top;
}

int Rectangle::area()
{
  return (x2-x1)*(y2-y1);
}
```

The "Rectangle::" notation at the beginning of the method definitions is necessary to specify the class that the methods belong to. As it is possible to have a number of other classes with an area() method also, e.g. OtherClass::area(), the notation differentiates between them.

Once the Rectangle class is defined, a program that uses it may be written. The class declaration must appear before main(). The actual method definitions may come before or after main().

OOP C++ Program to Calculate Area of Rectangle

```
void main()
{
  Rectangle R = Rectangle(12, 23, 34, 45);
  cout << "Area =" << R.area() << '\n';
}
```

Although there is some extra work to design and define the Rectangle class, the program becomes simpler once the class is available. It may also be used in other programs. Use of each Rectangle object needs:

- 1 more variable (an object of type Rectangle)
- 1 initialization statement (using constructor)
- 1 simple method call

Methods are called using a dot notation. In OOP, calling a method of an object is called "sending a message to the object". In this case, it is a "give me your area" message, i.e. main() is asking the Rectangle R to calculate its own area. The general form of a message is:

objectname.methodname(arguments if any)

Note that the call to area() does not need any arguments. Since the code in area() used by R can access the data members that belong to R, no arguments are passed.

Each object of the Rectangle class has its own set of data members, so calling the area() method for different Rectangle objects will return their respective areas.

OOP C++ Program for Area of Two Rectangles

```
void main()
{
  Rectangle R1 = Rectangle(12, 23, 34, 45);
  Rectangle R2 = Rectangle(10, 10, 100, 100);
  cout << "Area of R1=" << R1.area() << '\n';
  cout << "Area of R2=" << R2.area() << '\n';
}
```

Once the Rectangle class is defined, one can add more functionality to it by adding useful methods with minimal alteration of existing code. Thus, OOP software can be easily extended.

For example, if the circumference of a Rectangle object is required, one can easily write an additional method for the Rectangle class to return the required value. Additional code consists of:

- **A method prototype must be added inside the class definition:**

```
int circum();
```

- **The circum method can be defined outside the class definition:**

```
int Rectangle::circum()
{
  return 2 * ( (x2-x1) + (y2-y1) );
}
```

- **In main(), the following statement can then be added:**

```
cout << "Circumference =" << R.circum() << '\n';
```

Data Guarding

One important advantage of OOP is that since the data for an object is kept inside that object and is normally made invisible (hidden) to code outside the object, it can be guaranteed that the data remains valid and consistent by making checks in the code of its methods.

For example, if it is required that the initialization must ensure that the Rectangle coordinates are always set so that x2>x1 and y2>y1, the constructor may be re-written to initialize the data to valid values no matter what the passed-in values were:

Constructor with Data Guarding

```
Rectangle::Rectangle(int left, int bot, int right, int top)
{
  if ( left < right ) {
     x1 = left; x2 = right; }
  else {
     x2 = left; x1 = right; }
  if ( bot < top ) {
     y1 = bot; y2 = top; }
  else {
     y2 = bot; y1 = top; }
}
```

With the constructor code as written above, it is impossible for the area() method to return a negative value, as it is thus guaranteed that x2>x1 and y2>y1. Since the data members are hidden, they can never become invalid.

As the data members of a class are hidden and guarded, they can only be obtained or altered if required by providing some appropriate access methods for those purposes.

Access Methods

Data members and methods can be:

- Public: visible and accessible to code outside the object, i.e. to code that is not in any of the object's methods.
- Private: invisible and inaccessible to code outside the object.

In order to allow for altering the values of an object's private data members, public methods are needed for passing such messages. Code of those methods must also ensure that invalid values are not allowed.

Example: Method prototypes for resizing and shifting the rectangle:

```
void resize( double xfactor, double yfactor);
void shift( int xdir, int ydir);
```

Access Methods for Altering Values of Data Members

```
void Rectangle::resize( double xfactor, double yfactor) {
  x2 = (x2-x1) * abs(xfactor) + x1;
  y2 = (y2-y1) * abs(yfactor) + y1;
}

void Rectangle::shift( int xdir, int ydir )
{
  // negative coordinates are not allowed

  if ( x1+xdir >= 0 ) {
     x1 = x1 + xdir;  x2 = x2 + xdir; }
  else {
     x2 = x2 - x1;  x1 = 0; }

  if ( y1+ydir >= 0 ) {
     y1 = y1 + ydir;  y2 = y2 + ydir; }
  else {
     y2 = y2 - y1;  y1 = 0; }
}
```

Access methods may also be written to obtain the values of an object's private data members. Example:

```
int getLeft();
int getBot();
int getRight();
int getTop();
```

Access Methods for Obtaining Values of Data Members

```
int Rectangle::getLeft() {
  return x1;
}

int Rectangle::getTop() {
  return y2;
}
```

An example of a main program that uses the additional methods given:

Main Program Using Access Methods

```
void main()
{
  Rectangle R = Rectangle( 12, 23, 34, 15 );
  cout <<"Initial rectangle:\n";
  cout <<"Pt 1: ("<<R.getLeft()<<","<<R.getBot()<<")\n";
  cout <<"Pt 2: ("<<R.getRight()<<","<<R.getTop()<<")\n";

  R.resize( 2 , 0.5 );
  R.shift( -20 , 10 );

  cout <<"Final rectangle:\n";
  cout <<"Pt 1: ("<<R.getLeft()<<","<<R.getBot()<<")\n";
  cout <<"Pt 2: ("<<R.getRight()<<","<<R.getTop()<<")\n";
}
```

Method Overloading

It is sometimes convenient to have different versions of a method that do the same thing, i.e. several methods with the same name, but which have different arguments.

Overloading constructors is very common, in order to provide several ways of creating an object of the same class. As an example, recall the constructor for the Rectangle class that takes 4 integer arguments for the coordinates. Other constructors may be provided for creating some rectangle objects with different arguments. Prototypes can be:

```
Rectangle(int, int, int, int);  // 4 coordinates
Rectangle(int, int);            // point 1 at (0,0)
Rectangle(Point, Point);        // using Point struct
```

Overloaded Constructors

```
Rectangle::Rectangle(int left, int bot, int right, int top)
{
  x1 = left; x2 = right;
  y1 = bot; y2 = top;
}

Rectangle::Rectangle(int right, int top) {
  x1 = 0; x2 = right;
  y1 = 0; y2 = top;
}

struct Point {
  int x, y;
};

Rectangle::Rectangle(Point pt1, Point pt2) {
  x1 = pt1.x; x2 = pt2.x;
  y1 = pt1.y; y2 = pt2.y;
}
```

The rule for creating different methods with the same name is that the number or type of arguments must be different in each case, so the compiler can tell which one to call. The return value does not count.

It is possible to include the following overloaded method prototypes in a class definition:

```
void act();
void act(int);
void act(int, int);
int act(int, double);
void act(double, int);
```

It is incorrect to add the following prototype to the above, because the compiler cannot distinguish the correct one to use:

```
int act(int, int);
```

An informal rule is that functions with the same name should do the same thing. It is possible to have the following method prototypes for two different uses:

```
void length(int);
int length();
```

The first can be used to set the length of an object, e.g.:

```
obj.length(10);
```

The second can be used to get the length of the object, e.g.:

```
int len = obj.length();
```

Although they both are related to length, they do very different things, so this is not a recommended use of the overloading feature.

<u>Tutorial:</u> Write definitions of a Circle class with its methods:

- **Constructors for given int center coordinates and radius, int radius with center at (0,0), and double radius with center at (0,0).**
- **Methods for obtaining area and circumference.**
- **Methods for getting the values of the data members.**
- **Methods for resizing and shifting.**
- **Method for printing values of a Circle object in a clear form.**