CLASSES : defining, subclasses, methods, constructors, ...

Classes are the building blocks of C++. However, they are really nothing more than souped-up structures. Data within a class is considered private unless specifically identified as public (or protected). Private data is used only by member-functions within a class, otherwise known as methods.

CLASS DEFINITION

```
class Name:storage-specifier

   private:
      member-declarations
   protected:
      member-declarations
   public:
      member-declarations
;
```

Optional storage-specifier can be either direct or indirect. Default access-specifier is private (by comparison, all data in a structure is public).

SUBCLASS DEFINITION

Subclasses inherit the functionality of the parent class. If a method within the subclass has the same name as a method in the parent (or superclass), this method is said to override that of the superclass.

```
class Name:access-specifier Superclass

   private:
      member-declarations
   protected:
      member-declarations
   public:
      member-declarations
;
```

The access-specifier can be either public or private (default). If public, the public and protected members of the superclass are treated the same in the subclass. If private, the public and protected members of the superclass are treated as private members of the subclass.

DEFINING METHODS

Methods are functions which use and manipulate data within a class, and perform functions based on that data.

```
void className::methodName( arguments )

   declarations
```

```
    statements
```

CONSTRUCTORS AND DESTRUCTORS

Constructors and destructors allow you to initialized things when a new class is created, and clean-up house when your done. When memory is allocated for a class (when using the 'new' keyword for example), a class constructor is automatically called. A constructor has the same name as the class. You can define multiple constructors for a class, but each one must have a unique set of arguments. When a class variable goes out of scope, or is removed using the 'delete' keyword, the class destructor is called.

```
  class Name

    public:
      Name( int );        // Constructors have the same name as the class.
      ~Name();              // Destructor name uses ~ (only one allowed).
  ;

  Name::Name()              // Constructor code.

    declarations
    statements
  ;

  Name::Name( int a )    // Constructor code.

    declarations
    statements
  ;

  Name::~Name()             // Destructor code (no arguments).

    declarations
    statements
  ;
```

VIRTUAL METHODS

You define virtual methods by preceding the declaration with the 'virtual' keyword. Most base classes should use the 'virtual' keyword in front of all of it's member functions (or methods).

```
  virtual void myMethod1();         // virtual function
  virtual void myMethod2() = 0;   // pure virtual function
```

You cannot define any objects for a class that contains a pure virtual function and the class is said to become an 'abstract' base class.

EXAMPLE USING CLASSES WITH VIRTUAL METHODS

The following snippet shows how to define classes, subclasses, and methods (member functions) within classes. The important aspect of classes and C++ is that subclasses inherit (or override) the instance variables and methods of their parent or superclass (which in turn inherits the capabilities of it's superclass or... grandparent?). Most base classes use the virtual keyword for method declarations to allow for 'cleaner' inheritance of class methods. If your using pointers to objects (i.e., an array of objects), the compiler may not be able to resolve what type of object your sending a message to unless you use the virtual keyword in the base class. Identifying methods as virtual adds some more house-keeping which the compiler must perform, but it assures that overrided methods are handled properly in all cases.

```
// virtual.cp
#include <iostream.h>
#include<string.h>

const int kFourWheelDrive = 1;
const int kTwoWheelDrive   = 0;

class Vehicle

      char *make;
      int   year;
   public:
      Vehicle( char *m, int y );
      ~Vehicle( void )  delete make;
      virtual void print( void );
;

Vehicle::Vehicle( char *m, int y )

   make = new char[ strlen(m)+1];
   strcpy( make, m );
   year = y;


void Vehicle::print( void )

   cout << "Make: " << make << endl;
   cout << "Year: " << year << endl;


class Truck:public Vehicle

      int id;
   public:
      Truck( int i, char *, int );
      ~Truck( void )
      virtual void print( void );   // virtual keyword not needed, but
                                       // is usually added for clarity.
;

Truck::Truck( int i, char *m, int y ):Vehicle(m,y)

   id = i;


void Truck::print( void )

   Vehicle::print();                  // call Vehicle print method
   cout << "Type: ";
   if( id )
```

```
      cout << "4x4" << endl;
   else
      cout << "4x2" << endl;


void main( void )

   Vehicle crx( "Honda", 1985 );
   Truck    ranger( kFourWheelDrive, "Ford", 1993 );

   Vehicle *ptr[2];   // you can store pointers to Vehicle *or* Truck
                      // objects into this array (or pointers to *any*
                      // sub-class of Vehicle for that matter).

   ptr[0] = &crx;
   ptr[1] = &ranger;

   for( int counter = 0; counter < 2; counter++ )

      // following statement requires virtual method!
      ptr[counter]->print();
      // in above statement, compiler doesn't know beforehand which
      // print method to call.   Use of virtual keyword in base class
      // allows compiler to determine this on-the-fly.

      cout << endl;

      // try deleting 'virtual' keyword in Vehicle class
      // and see what happens.


// end virtual.cp
```