Object-Oriented Programming

More Class Features

More class features

- composition: objects within objects
- accessor, mutator and utility member functions
- The **this** pointer
- static members
- friend functions

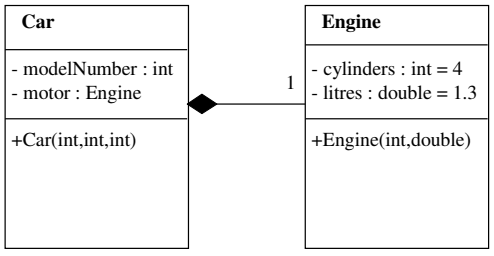
Object Composition

- Data members of a class can be objects of another class
- New classes can be *composed* of objects of existing classes
- Object composition is one of the most powerful tools in OOP
- Composition within OOP leads to more code re-use

Examples of Object Composition

- The data members of an Appointment class
  - name – a string indicating who made the appointment
  - appDate – a Date object for the day
  - appTime – a Time object for the time
- The data members of a Car class
  - modelNumber – an integer showing the model
  - motor – an Engine object describing the car's engine

UML diagrams for Car



UML diagrams for Car

- UML composition.
  - Each engine belongs to only one car
  - The engine lives and dies with the car
  - Destroying the whole (car) destroys the part (engine)
  - Car is the *client class*. Engine is the *supplier class*

Object composition in C++

- Very similar to object composition in UML
  - Each included object is associated with only one whole object
  - The included object is created at the same time as the whole
  - The included object is destroyed at the same time as the whole


```
class Engine
{
 public:
   // Constructor
   Engine(int c = 4, double l = 1.3);
   // ...
 private:
    int cylinders; // Number of cylinders
    double litres; // Capacity in litres
};
```



```
// Implementation file for the Engine class

// Constructor
Engine::Engine(int c, double l) : cylinders(c), litres(l)
{
}
// ...
```



```
class Car
{
public:
    // Constructor
    Car(int m, int c, double l);
    // ...
private:
     int modelNumber;    // Model number
     Engine motor;       // Car's engine
};
```


Implementation file

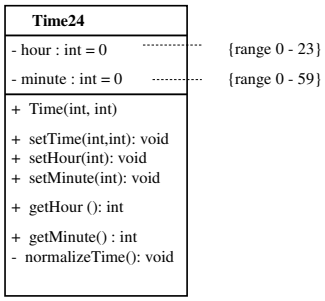
```
// Constructor. The data member motor is
// initialised using the values of the parameters // c and l.

Car::Car(int m, int c, double l) :
        modelNumber(m), motor(c, l)
{
 }
 // ...
```


Accessor, mutator & utility member functions

- Accessor functions provide access to the object's private data members
- Mutator functions allow the user to alter private data members
- Utility functions are (usually) private functions that perform housekeeping duties

Time24 class




```
void Time24::normalizeTime ()
// utility function to ensure valid time
{
    int extraHours = minute/60;

   // minute in range 0 to 59
   minute = minute%60;

   // hour in range 0 to 23
   hour = (hour+extraHours)%24;
}
```

normalizeTime (utility function)


```
Time24::Time24 (int h, int m) : hour(h), minute(m)
// use the utility function to ensure valid time
{
    normalizeTime();
}

Time24::setTime(int h, int m)
{
   setHour(h);
   setMinute(m);
}
```



```
void Time24::setHour(int h)
{
   hour = ( h>=0 && h<24) ? h : 0;
}
void Time24::setMinute(int m)
{
   minute  = ( m>=0 && m<60) ? m : 0;
}
```



```
int Time24::getHour()
{
   return hour;
}
int Time24::getMinute()
{
   return minute;
}
```


Constants in C++

- Value substitution – naming frequently used constant values

```
const int ARRAY_SIZE = 40;
int myArray [ARRAY_SIZE];
```

- For safety – when the value of an object (or variable) will not change during its lifetime

Const and Classes

- Member functions that do not modify member data should be declared **const**
- **const** must be added to both the function prototype and implementation

```
int getHour()  const;
int Time24:: getHour () const
{
   ……
}
```


const Objects

- Objects are seldom passed by value in C++. If this functionality is required, they are passed by **const** reference

  void displayTime(const Time &t)

- Passing by **const** reference
  - Avoids copying the object
  - Protects the object from unintentional changes. (principle of least privilege)

Static data members

- A data member that is declared static is shared by all instances of a class. Only one copy of the member exists.

```
// Header file for Earthquake class

class Earthquake
{
public:
  static int numberOfQuakes;          // Number of earthquakes
  // Constructor
  Earthquake();
  // ...
private:
  double magnitude;          // Measurement on Richter scale
};
```


Defining static data members

- The definition of a static data member within a class does not allocate storage for that member. To allocate space for a static data member, you must make an additional global declaration outside the class header.

```
// Implementation file for the Earthquake class, quake.cpp
int Earthquake::numberOfQuakes = 0;
// Constructor
Earthquake::Earthquake()
{
  ++numberOfQuakes;
}
// ...
```


Accessing static data members

- A member function can refer to a static data member of the same class directly.
- A non-member function can refer to a static data member using either the notation

  ClassName::staticMemberName **or**
  objectName.staticMemberName


```
#include <iostream.h>

int main()
{
  Earthquake sanFrancisco89;
  cout << Earthquake::numberOfQuakes << endl;
  // ...
  Earthquake losAngeles94;
  cout << losAngeles94.numberOfQuakes << endl;
  // ...
  return 0;
}
```

Accessing static data members

Static member functions

- Like static data members, *static member functions* are associated with a class and not with any particular object of that class.
- Static member functions do not have a this pointer.
- Static member functions cannot access non-static members of their class.
- Static member functions cannot be declared const.

Accessing static member functions

- A member function can call a static member function of the same class directly.
- A non-member function can call a static member function using either the notation

ClassName::staticMemberName() or
objectName.staticMemberName()


```
class Earthquake
{
 public:
   // Constructor
   Earthquake();
  // Returns the number of earthquake measurements
  static int getNumberOfQuakes();
  // ...
private:
   double magnitude;   // Measurement on Richter scale
   static int numberOfQuakes; // Number of earthquakes
};
```



```
// Implementation file for the Earthquake class
int Earthquake::numberOfQuakes = 0;
// Constructor
Earthquake::Earthquake()
{
  ++numberOfQuakes;
}
// Returns the number of earthquake measurements
int Earthquake::getNumberOfQuakes()
{
  return numberOfQuakes;
}
// ...
```



```
#include <iostream.h>
int main()
{
   Earthquake sanFrancisco89;
   cout << Earthquake::getNumberOfQuakes() << endl;
   // ...
   Earthquake losAngeles94;
   cout << losAngeles94.getNumberOfQuakes() << endl;
  // ...
  return 0;
}
```


The *this* pointer

- Whenever a member function is called, it is automatically passed a pointer variable named this, whose value is the address of the object that generated the call.
- The this pointer is an implicit argument, which means that it doesn't appear in the member function's parameter list.

```
// Header file for Rectangle class

class Rectangle
{
 public:
   // Returns the area of the rectangle
   double getArea();
   // ...
 private:
   double length;   // Length of rectangle
   double width;    // Width of rectangle
};
```



```
// Implementation file for the Rectangle class,
   rectangle.cpp

// Returns the area of the rectangle
double Rectangle::getArea()
{
   return (length * width);
}
// ...
```


Rectangle implementation

Inside getArea(), the statement

return (length * width);

could also be written as

return (this->length * this->width);

because this points to the object that invoked getArea().

Friend functions

- A *friend function* is a non-member function that has direct access to all private and protected members of a certain class.
- To make a function a friend of a class, place the function's declaration inside the class definition and precede it with the keyword friend.


```
// Header file for Rectangle class
class Rectangle
{
public:
   // Constructor
   Rectangle(double l, double w);

   // Returns the area of the rectangle
   friend double getArea(const Rectangle& r);
   // ...
private:
   double length; // Length of rectangle
   double width; // Width of rectangle
};
```



```
// Implementation file for the Rectangle class
// Constructor
Rectangle::Rectangle(double l, double w) : length(l), width(w)
{
}

// Returns the area of the rectangle
double getArea(const Rectangle& r)
{
  return (r.length * r.width);
}
// ...
```

Friend function rectangle.cpp

```
#include <iostream.h>
int main(void)
{
   Rectangle boundary(15.5, 20.0);
   cout << "Area is " << getArea(boundary) << endl;
   return 0;
 }
```

Using the Friend function

Friends of multiple classes

- A single function can be a friend of two or more classes simultaneously

```
#include <iostream.h>
class Warning;    // Forward class declaration
class Error
{
public:
   // Returns true (1) if either message is visible
   friend int visible(const Error& e, const Warning& w);
   // ...
private:
   int appearance;   // Value is hidden (0) or visible (1)
   // ...
};
```



```
class Warning
{
 public:
   // Returns true (1) if either message is visible
   friend int visible(const Error& e, const Warning& w);
   // ...
private:
    int appearance;    // Value is hidden (0) or visible (1)
    // ...
 };
```



```
// Returns true (1) if either message is visible
int visible(const Error& e, const Warning& w)
{
    if (e.appearance || w.appearance)
       return 1;
    else
       return 0;
}
```

visible.cpp


```
int main(void)
{
    Error errorMessage;
    Warning warningMessage;
    // ...
    if (visible(errorMessage, warningMessage))
       std::cout << "Screen is not clear." << std::endl;
    // ...
    return 0;
}
```

main.cpp

Member functions as friends

- A member function of one class can be a friend function of another class.
- When a member function is declared as a friend, the scope resolution operator must be used to qualify its name.