Classes in C++

Section 2

From Structures to Objects

In C and other procedural programming languages:

- •programming is action oriented
- •unit of programming is the function (or procedure)
- •data exists to support the actions that the functions perform
- •dynamically, functions are called

In C++ and other OO languages:

- •programming is object oriented
- •unit of programming is the class (a user-defined type)
- •data and functions are contained within a class
- •dynamically, objects are created (and used through their methods)

Object – Orientation

- **Above the level of data-structures and algorithms**
- **A means of organising programs by realising it's entities in an abstract manner**
- **A way of hiding an entities complexity**
- **A method of re-using code using entities as building blocks for new related entities**

Abstract Data Types

- We need to define
  - The abstract 'thing' we are trying to represent in our programs
  - The data representing the state of that thing
  - The behaviour of that thing
- The 'thing' we are trying to define is not the object, but a class of objects.
- The class defines the common attributes and methods for all objects of the class.

- Attributes: the elements which make up an objects state.
- In the class definition we are only concerned with an attributes name and type.
- Methods: the functions to change the attributes values.

C++ Syntax:

- **Class** defines an ADT
- **Private** defines the hidden part
- **Public** defines the visible interface of the class

- **Objects**
  - **Association of data with it's operations**
  - An integer has a set of associated operations
- **The Black Box**
  - No need to care about it's internal mechanics
  - **Hiding an entities constituent data**
  - The data is irrelevant to the outside
- **Communication via contract**
  - A specification of what it does

The Object Paradigm

- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Encapsulation

- Animating data
- Hiding Data from the client
- Moving away from manipulation of data by the user
- Interaction through an interface
- Communication between objects via messages
- Client Server approach to software components

Inheritance

- Re-use of existing objects
- Sharing similar behaviour
- New entities using facilities provided by existing ones
- Hierarchical approach

Objects in C++

- Extension to user-defined types
- Binds two familiar concepts
- Organising data into capsules
- Manipulation of data with functions
- Fusion being the difference

From Structures to Objects

Classes in C++ are a natural evolution of the C notion of **struct**

A typical structure example:

```
struct Time {
int hour;  // 0-23
int minute; // 0-59
int second; // 0-59
};
```

By exposing weaknesses of this struct example (which are typical of all structures) we will motivate the evolution towards classes

Using Structures … some revision

Structure variables are declared like variables of other types:

```
Time timeobj, timeArray[ 10], *timePtr,
```

We will not worry about reference declarations like:

```
&timeref = timeobj
```

Members of a structure (and a class) are accessed using:

dot operator: **.**

arrow operator: **->**

For example:

```
cout << timeobj.hour;
timePtr = &timeObject;
cout << timePtr->hour;
```

Using Structures … a word of warning

The arrow operator requires some care:

timePtr->hour is equivalent to **(*timePtr).hour**

Parentheses are needed because the dot operator (.) has a higher precedence than the pointer dereferencing operator (*)

Writing ***timePtr.hour** is a syntax error because the precedence rules define it as ***(timePtr.hour)** … since with a pointer you must use the arrow operator to refer to a member!

NOTE: pointers are difficult for beginners and experts! (esp. C++)

NOTE: (the concept of) pointers (is)are fundamental to (most) programming

Using Structures … with a function

With the Time structure a typical function would be to print out the time on the screen:

```
void print (Time t) \\ Just hours and minutes
{ cout << (t.hour < 10 ? "0": "") << t.hour <<":"
        << t.minute <10 ? "0": "") << t.minute;
}
```

Notes:

- structures are normally call-by-value
- could pass a constant reference (or pointer)
- the **test? True-do-this: False-do-this** construct

Using Structures … with a function… a problem?

We can now write the following:

```
Time t1;
t.hour = 66; t.minute = 78;  t.second = 100;
print (t1);
Time t2;
print (t2);
```

Here, t1 is initialised incorrectly and t2 is not initialised at all.

Also, if the implementation of the structure is changed (for example, if we just count seconds since midnight and calculate hour and minute values) then the print function (and all other functions acting on Time) must be changed accordingly… in fact, all programs using Time must be changed!

Using Structures … more (C) problems

In C (and C++), structures are treated member-by-member:

- structures cannot be printed as a unit
- structures cannot be compared in their entirety
- structures cannot have operators overloaded (like =, <, +, <<)

Moving from C structures to C++ classes is easy:

- classes and structures can be used almost identically in C++
- difference between the two is in accessibility to members

Classes as an ADT implementation

Classes let the programmer model objects which have:

- attributes (represented as data members), and
- behaviour (operations represented as member functions)

Types containing data members and member functions are defined in C++ using the keyword **class**

Consider **Time** as a class:

```
class Time {                    private:
public:                         int hour;  // 0 -23
Time ();                        int minute; // 0-59
void setTime(int, int, int);    int second; // 0 -59
void print()                    };
```


Class Time continued ...

```
class Time {                    private:
public:                         int hour;  // 0 -23
Time ();                        int minute; // 0-59
void setTime(int, int, int);    int second; // 0 -59
void print()                    };
```

- **public:** and **private:** are member access specifiers (ignore protected, for now)
- **public** --- member is accessible wherever the program has a **Time** object
- **private** --- accessible only to member functions of the **Time** class
- the class definition contains *prototypes* for the 3 public member functions.
- the function with the same name as the class is a *constructor*

Constructors

Constructors are fundamental to all OO programming:

- A special function used to initialise data members
- Called automatically when an object of the class is created
- Common to have several constructors for a class (overloading)
- Most OO languages (including C++) have default constructors
- Things get complicated when:
  - member objects of a class are themselves objects!
  - we inherit behaviour …
  - we have recursive data structures in classes

Time Constructor

Constructors can guarantee *valid* initialisation:

```
Time::Time() {hour = minute = second =0;)
```

Methods can guarantee *valid* state changes:

```
void Time::setTime (int h, int m, int s){
hour = (h>=0 && h<24) ?h:0;
minute = (m>=0 && m<60) ? m:0;
second = (s>=0 && s<60) ? s:0;
```

Maintaining these types of *invariant property* is a good programming style

NOTE: member functions can be defined inside a class, but it is good practice to do it outside the class definition

NOTE: The C++ scope resolution operator (**::**)

Defining Object Types

- **User-defined types defined using struct**
- **Object types defined using keyword class**

```
class ObjectType
{
  // Declarations
};
```

- **Like structs classes contain data variables (known as attributes/member variables)**


```
class Counter
{
  int Count;
};
```

Classes also contain functions (known as member functions or methods)

```
class Counter
{   int Count;
  void Increment();          };
```

Classes are declared in header files

Instantiating Objects

- Objects are instances of classes
- This follows from instances of user-defined types where a struct is a template for an instantiation
- To instantiate an object of the Counter class

```
void Func()
{
 Counter count1;
}
```


- **As with structs pointers may be used to reference dynamic Objects**

```
void Func()
{
 Counter *pCount = new Counter;
}
```

Accessing class members

- **By default class members cannot be accessed as can fields in a struct**
- For the above count class the following would be illegal:

```
void func()
{
 Counter count1;
 count1.Count = 0;
}
```

- The rules of encapsulation do not allow member variables to be accessed directly from outside an object

- The rules of encapsulation do not allow member variables to be accessed directly from outside an object
- It is possible though to allow members to be visible from outside an object

Access Modifiers

- **C++ classes provide access modifiers which specify where a member is visible**
- **There are three access modifiers**
  - **private**
  - **public**
  - **protected** (We will ignore this for the time being)

Private v Public

- The private access modifier states that the members following it are visible only within that class (This is the default)
- The public access modifier states that the members following it are visible both inside and outside that class

Using Access Modifiers

- If we want to allow all the members of the count class to be visible outside the class we would place the public access modifier at the beginning of the declarations for the class

```
class Counter
{
public:
  int Count;

  void Increment();
};
```


- This breaks the rules of encapsulation though we might want only the Increment method visible externally

```
class Counter
{
private:
   int Count;
public:
  void Increment();
}
```


Methods

- The Counter example contains what appears to be a function prototype called Increment
- This is what is known as a Method or a member function
- A method cannot be used alone because it is associated with a class
- A method is referenced similarly to a member variable using '`.`' or '`->`' (Depending on whether an instance or pointer is used)


```
void func()
{   Counter *pCount = new Counter;
  PCount->Increment();
}
```

- By referencing a method in this manner we are said to have invoked it
- A method is invoked on an Object
- Methods are like funcitons in many respects
  - Can return values
  - Can take parameters

- Scope resolution operator i.e. **::** is used to link together classes and definitions of their methods.
- It is a normal practice to put attributes into the private part of the class, where they can only be accessed via methods.

- Methods themselves appear in the public part of the class so that they may be accessed externally and provide an interface to the class.
- It is possible to put attributes into the public part of the class but this breaks the rules of encapsulation.
- It is also possible to put methods in the private part of the class. This is useful for methods which are used by other methods of the same class, but not appropriate as part of the external interface.

- Inline methods are declared and defined within the body of the class => duplicated for every object of the class.
- Usually methods are declared within a class and defined outside the class

Implementing Methods

- It is apparent that there is no implementation for the Increment method
- As with functions the implementation for methods is kept in a source file
- A method implementation declaration includes the class name and the method name seperated by **‘::’**

```
void Counter::Increment()
{Count++;}
```

- All other class members are accessible directly from within a method
- The public methods are usually the interface to a class. They set up the contract between the class and its user.

An Example.

```
class Example
{
 private:
   int i;

   public:
        void setvalue (int value);
        int get_value();
};   // note ;
```


The Method Definitions

```
void Example :: setvalue(int value)
{
   i= value;
}  // note absence of ;

int Example::get_value()
{
   return i;
} // note absence of ;
```


- The public methods are the only things which can be used with an object of a class. There is no direct access to the private elements.
- Calling a method is known as sending a message.
- Any instantiated object can be sent messages using the dot operator e.g.
  - **classname.methodname(parameters)**
- Example.setvalue(10);

  Sets i to the value 10.

- Where we define a class we store it in a header file e.g. Example.h.
- This file is included in program by
  #include “Example.h”
- Note <Example.h> = > look for file in the usual library directory.
- ***A class defines the types of data appropriate to the class (the attributes) and its set of allowed behaviours (the methods).***

- Class = “Object Factory”
  – i.e. classes allow us to create new objects of the class, each one of which follows an identical pattern of attributes and methods.
- These objects are declared within main.

Given the declaration of a class called Bank_Account stored in the file Bank.h, the following code allows us to use the class Bank_Account to

- create an object MyAccount
- credit the bank account with 20 pounds, using the method credit from the Bank_Account class.

```
#include ”Bank.h”
void main()
{ Bank_Account MyAccount;
 Bank_Account.credit(20);
}
```


Class v Object

- Classes:
  – Unique name, Attributes, Methods
- Object:
  – Identity, State, Behaviour
- Classes exist all the time when a program is running whereas Objects may be created and destroyed at run time.
- Any number of objects may be instantiated for a given class.

Initialising An Object

We have seen before that it is possible to initialise variables of an intrinsic type using assignment `int x = 0;`

- Because an object may be quite complex a mere assignment may not be possible
- To allow objects to be initialised a special method called a constructor is used

Constructors

- Used to allocate memory for an object. Constructors are automatically called when an object is created.
- An object constructor
  - takes the same name as the class
  - may have arguments
  - cannot return a value.

- Constructors are executed every time an Object of a particular class is created
- This method takes the same name as the class and does not return a value

```
class Counter
{private:
        int Count;
 public:
        Counter(); // Constructor
        void Increment();
};
```


The 3 types of Constructor

- The default constructor – takes no parameters, performs no processing other that the reservation of memory . The compiler always calls it if no user-defined constructor is available.
- User defined constructors – used to initialize an object when it is created. Over loading is possible.
- The copy constructor – a way of using assignment when creating new objects.
  - New_object object2 = object1
  - This instantiates object 2 as object 1 has been instantiated.

User Defined Constructor

For user defined constructor definitions look as follows:

```
Bank_Account:: Bank_Account()
{   current_balance = 0
    // Initialize attributes here.}
```

In main an object will be set up as follows:

Bank_Account MyAccount();

giving a bank account with the current balance of 0.

User Defined Constructor

- It is also possible to pass parameters to a constructor.

```
Bank_Account:: Bank_Account(int x)
{
   current_balance = x
   // Initialize attributes here.
}
```

In main an object will be set up as follows:

```
Bank_Account MyAccount(100);
```

giving a bank account with the current balance of 100.

Class Declaration for the Counter Class

```
class Counter
 {
 private:
            int Count;

 public:
            Counter();
            int GetCount();
            void Increment();
   };
```


Body for Counter Class

```
#include "Countr.h"
Counter::Counter()
{
 Count = 0;
}
void Counter::Increment()
{
 Count++;
}
int Counter::GetCount()
{
 return Count;
}
```

Using the Counter Class

```
#include "Countr.h"
#include <iostream.h>
void main()
{
 Counter count1;
 while (count1.GetCount() < 10)
  {
   cout << " count ";
   cout << count1.GetCount();
   cout << endl;

   count.Increment();
  }
}
```


Destructors

A function with the same name as the class but preceded with a tilde character (~) is called the destructor

When not explicitly given (as in **Time**), the compiler provides a default

Destructors cannot take arguments … so are never overloaded

Destructors do termination housekeeping on each object before the memory is reclaimed by the system

These are very important when using dynamic data structures.