An Introduction to

Object Oriented Programming with C++

Start with a quick look at the most important features of the C++ language.
All of these will be covered in more detail as we move through the course.

A programming Language provides . . .

a vehicle for the programmer to specify actions to be executed and

a set of concepts for the programmer to use when thinking about what can be done

The first requires a language that is

"close to the machine", so that all important aspects of the machine are handled:

simply and efficiently, and

in a way that is reasonably obvious to the programmer.

The second requires a language that is

"close to the problem to be solved", so that concepts of a solution can be expressed:

directly, and

concisely.

Bjarne Stroustrup,

"The C++ Programming Language", 2nd Edition, 1994

Basic Aims of Object Oriented Programming

- Analysis, design and coding.
  In conventional (procedural) systems, there is a large semantic gap between these phases.
  Classes correspond to real-world objects, so that code structure closely relates to user requirements

- Separate development of parts of an application.
  Once the public interface of a class has been agreed on the development of code to support this class can proceed independently.

- Enhancement of an application.
  Polymorphism, or *dynamic binding*, of functions allows the addition of new sub-types of objects with minimal changes to existing code

Basic Aims of Object Oriented Programming

- Automatic data initialisation and clean-up.

  Class *constructors* and *destructors* allow the programmer to specify, in one place, how objects of that class are to be initialised and destroyed.

- Re-use of existing code.

  Inheritance allows existing code to be tailored to a new application without access to its source code.

Class

- This is the basic concept in Object Oriented program development

- Most classes correspond to a type of “thing” in the problem domain, e.g.
  Person,
  Savings account,
  Transaction, etc.

- Class encapsulates:
  The type of information to be held about every object of that class
  (*attributes*, or *data members*).
  The types of operations that may be performed on these objects
  (*behaviour* or *member functions*).

- Enhancement of an application.
  Polymorphism, or *dynamic binding* of functions, allows the addition of new types of objects with minimal changes to existing code.

Declaration of a Class

In "person.h"

```
#ifndef PERSON_H
#define PERSON_H
#include <string>
using namespace std;
class Person
{
   public:
                     Person(string nameIn, int ageIn);
                     ~Person();
              void   GrowOlder();
              int    GetAge();
              string GetName();
   private:
              int    age;
              string name;
};
#endif
```

Implementation of a Class

In "person.cpp"

```
#include "person.h"

Person::Person(string nameIn, int ageIn){
     name = nameIn;
     age  = ageIn;
}

Person:: ~Person(){};

void Person::GrowOlder(){
     age++;
}

int   Person::GetAge(){
     return age;
}

string Person::GetName(){
     return name;
}
```

Objects of a Class

- *Objects* of a class can be declared in a similar way to the declaration of objects of the built-in types such as *int*, *double* etc.
- Member functions can be applied to these objects.

e.g. in "main.cpp"

```
#include "person.h"
void main(){
   int w, x, y;
   Person john("John", 21); // john and jim are objects of type
   Person jim("Jim", 18);   //   Person. Constructors will
                            //   initialise name and age
                            //   with the values supplied.

   Person* pPtr;            // pPtr is pointer to a Person object
   pPtr = &jim;
   w = john.GetAge();       // w contains 21
   x = pPtr->GetAge();      // x contains 18
   john.GrowOlder();
   y = john.GetAge();       // y contains 22
};
```

Dynamic Allocation of Objects

- Objects of a class can be dynamically allocated. e.g. in “main.cpp”

```
#include ”person.h”
void main()
{
   int y, z;
   Person* pPtr;          // pPtr is a pointer to a Person

   pPtr = new Person(“Jill”, 45);
            // pPtr points to a dynamically allocated Person
            //   which has been initialised with  “Jill” and 45

   y = pPtr->GetAge();    // y contains 45
   pPtr->GrowOlder();
   z = pPtr->GetAge();    // z now contains 46
};
```

Class (cont'd)

- Objects can be variables in expressions –
  provided that the meaning of the operators being applied to them have been defined.

```
#include "person.h"

void main()
{
   Person jim("Jim", 22);
   Person john("John", 24);
```

```
   if(jim + 4 > john)
   {
      . . . . . .
   }
      // This is legal and meaningful once we have defined the
      //  meaning of "+" and ">" in this context.
```

Inheritance - public

- A new (*derived* or *child*) class is identical to another (*base* or *parent*) class except that:
  - We wish to hold additional information about objects of the derived class and/or
  - There are operations that are appropriate for the derived class but not for the base class

- The set of objects defined by the derived class is a subset of the base class objects.

- This is by far the most important form of inheritance.

- Identification of public inheritance relationships between classes is an important part of the analysis phase.

Inheritance – public (example)

- An Employee is a Person.
- All data we wish to hold about a Person, we must hold for an Employee.
- All operations appropriate to a Person are also appropriate for an Employee.
- For an Employee we need additional data such as:
  - Employee Number
  - Salary information, etc.
- For an Employee, additional operations are necessary such as:
  - Calculate pay

There is a clear indication that Employee class should
Inherit from
The Person class.

Inheritance – public (syntax)

```
          // in person.h
#ifndef PERSON_H
#define PERSON_H
#include <string>
class Person
{
   public:
          Person(
           char* naIn, int aIn);
          ~Person();
     int   GrowOlder();
     string GetName();
     int   GetAge();
   private:
     int   age;
     string name;
};
#endif
```

```
          // in employee.h
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include “person.h”
class Employee : public Person
{
   public:
          Employee(
           char* nIn, int aIn,
           int noIn, float sIn);
          ~ Employee();
     float CalcPay();
   private:
     int   empNo;
     float salary;
};
#endif
```

Inheritance – public

- Every object of the Employee class will contain the data members:
  - **`age`** (inherited from Person)
  - **`name`** (inherited from Person)
  - **`empNo`**
  - **`salary`**

- The following functions can be applied to any Employee object:
  - **`GrowOlder()`** (inherited from Person)
  - **`GetAge()`** (inherited from Person)
  - **`GetName()`** (inherited from Person)
  - **`CalcPay()`**

Inheritance – non-public

- An object of one class *bears some similarity* to objects of another class.
- Code and data structures from the base class may be useful in the implementation of the other class.
- In this case we may derive the new class from the existing class using non-public inheritance.

For example:

A dog is not a person, but

certain items such as

`name, age, address`

and member functions to manipulate these

are common to both classes.

Non-public inheritance may be useful in the implementation of a Dog class.

Polymorphism

- A variable that is declared to be *a pointer to a base class* may contain the address of:
  - An object of the base class, or
  - An object of any class which inherits publicly either directly or indirectly, from that class.

- A function may be declared to be *virtual* in the base class.
  - This function may be *over-ridden* in any of the child classes.

- If a virtual function is invoked via the pointer
  - *The version of the function that will be invoked depends on the exact class of the object addressed.*

The function to be called is not determined until run time.

Polymorphism – virtual functions

```
class Employee : public Person{
   public:
      virtual float CalcPay()=0;   // CalcPay() is a pure virtual
   private:                        //  function, hence Employee is
};                                 //  an abstract class
```

```
class Salaried : public
   Employee{
   public:
      float CalcPay(){
         return salary / 12;
      }
   private:
      float salary;
};
```

```
class Hourly : public Employee{
   public:
        float CalcPay(){
            return hrlyRate
              * hrsWorked;
        }
   private:
        float hrlyRate, hrsWorked;
}
```

Using Polymorphism

```
void main(){
   int noEmps = 0;
   Employee* employees[100];
    . . . .
   employees[noEmps++] = new Salaried(. . . .);
    . . . .
   employees[noEmps++] = new Hourly(. . . .);
    . . . .
   for(int i = 0; i < noEmps; i++){
      cout << “Name: “ << employees[i]->GetName()
         << “ $” << employees[i]->CalcPay() << endl;
}
```

- The version of **`CalcPay()`** to be executed depends on whether **`employees[i]`** is a Salaried or an Hourly.
- Addition of a new type of employee requires defining a new class.
- Minimal changes to existing code are required.