University of Essex

C++ Programming

10 – Inheritance, Abstract Classes and Virtual Functions

Inheritance

- Inheritance is a form of software reuse: we create a new class that absorbs the data and behaviours of an existing class and enhances them with new capabilities (new members, redefined members).
- The terms superclass and subclass are commonly used to describe the new and old class, but C++ uses the terms ***derived class*** and ***base class***.
- Although a derived class possesses all of the members of its base class the private members of the base class cannot be accessed directly in the methods or friend functions of the derived class;

Inheritance

- C++ provides three kinds of inheritance: public, protected and private.
- When using public inheritance the public and protected members of the base class are regarded as public and protected members of the derived class,
- When using protected inheritance the public and protected members of the base class are regarded as protected members of the derived class and when using private inheritance all members of the base class are regarded as private members of the derived class.

Inheritance

- The use of protected and private inheritance in C++ is relatively rare; public inheritance is required in most applications.
- If a class has a member with the same name (and in the case of a member function the same argument types) as a member of its base class, it will replace the inherited member.
- Data members with the same names should usually be avoided but it is quite reasonable to have methods with the same name,

Inheritance

- If a class A is a subclass of B, which is in turn a subclass of C then C is said to be an ***indirect*** base class of A whereas B is a ***direct*** base class of A.
- In C++ (unlike Java) multiple inheritance is allowed – a class may have more than one direct base class.
- For example the class **ifstream** has as base classes both **istream** and **fstream**.
- If two base classes have members with the same name it is necessary to avoid ambiguity – we would have to redefine the member in the derived class.

Inheritance

- The syntax used to indicate inheritance in C++ differs from that of Java; there is no **extends** keyword.
- We indicate that **Student** is a derived class of **Person** using a declaration of the form

```
class Student: public Person
{
    private:
        int year, regNo;
    public:
        Student(string name, int year, int regNo) : Person(name)
        {
            this->year = year; this->regNo = regNo;
        }
}
```

- The use of **public** indicates that public inheritance is being used.

Inheritance

- A constructor for a derived class must invoke the constructor(s) of its direct base class(es).
- This may be done explicitly as part of an initialiser list, as in the example on the previous slide (where we have assumed that the **Person** class has a one-argument constructor);
- otherwise the no-argument constructor from the base class (if it exists) will be invoked implicitly before execution of the body of the derived class constructor.
- If a constructor needs to explicitly initialise inherited members to values that differ from those that would be set by a base class constructor this must be done by assignment in the function body; inherited members cannot be initialised in an initialiser list.

Inheritance

- A destructor for a derived class will always implicitly invoke the destructor(s) of its direct base class(es);
- The body of a destructor written by the programmer will be executed before the base class destructor(s) .
- A call to a copy constructor for a base class may supply a derived class object as its argument. For example in a declaration such as **Person p(s);**
- (where **s** is an object of type **Student**) the **Person** object will be initialised to be a copy of the inherited attributes stored in the object **s**.
- The same applies to assignment operator: **p = s** will perform assignment using the inherited attributes.

University of Essex

Inheritance

- A member function of a derived class will sometimes need to invoke a member function of the base class that has been redefined in the derived class.
- In particular a method will often need to invoke the method that it replaces.
- For example if the **Student** class has a **print** member function this may wish to invoke the **print** function from the **Person** class to print the values of the inherited members.
- To invoke a method that has been redefined the call must be preceded by the name of the base class, followed by **::**.

Inheritance

```cpp
void Person::print(ostream &o) const
{
    o << "Name: " << name;
    // would normally expect to print some other
    // attributes as well
}
void Student::print(ostream &o) const
{
    Person::print(o);
    o << "; Year: " << year
    << "; Registration number: " << regNo;
}
```

Inheritance – an example

- We now present a detailed example of the use of inheritance.
- A commission employee earns only commission on the sales that he makes, whereas a base-plus-commission employee also earns a basic flat rate salary in addition to his commission on sales.
- We show on the following slides base class **CommissionEmployee** and a derived class **BasePlusCommissionEmployee** written with separate header files.

Inheritance – an Example

```
#ifndef _COMMISSION_H_
#define _COMMISSION_H_

#include <string>

using namespace std;

class CommissionEmployee
{
    public:
        CommissionEmployee(const string &, const string &,
            const string &, double = 0.0, double = 0.0);
        void setFirstName(const string &);
        string getFirstName() const;
        void setLastName(const string &);
        string getLastName() const;
        void setSocialSecurityNumber(const string &);
        string getSocialSecurityNumber() const;
        void setGrossSales(double);
        double getGrossSales() const;
        void setCommissionRate(double);
        double getCommissionRate()
        const; double earnings() const;
        void print() const; // prints object to stdout
    private: // could also use protected
        string firstName, lastName, socialSecurityNumber;
        double grossSales; //gross weekly sales
        double commissionRate; //commission percentage
}; #endif
```

University of Essex

Inheritance – an example

```
#include <iostream>
#include "CommissionEmployee.h"

CommissionEmployee::CommissionEmployee(const string &first,
      const string &last, const string &ssn,
      double sales, double rate): firstName(first), lastName(last)
{ // use member functions since validation needed
      setGrossSales(sales); setSocialSecurityNumber(ssn);
      setCommissionRate(rate);
}
double CommissionEmployee::earnings() const
{
      return commissionRate * grossSales;
}
// other member function definitions needed
// (set/get functions, print)
```

Inheritance – an example

```
// BasePlusCommissionEmployee.h
#ifndef _BASEPLUS_H_
#define _BASEPLUS_H_
#include <string>
#include "CommissionEmployee.h"

class BasePlusCommissionEmployee:
public CommissionEmployee
{
    public: BasePlusCommissionEmployee(const string &,
        const string &, const string &, double = 0.0,
        double = 0.0, double = 0.0);
        void setBaseSalary(double);
        double getBaseSalary() const;
        double earnings() const; // overrides inherited member
        void print() const; // overrides inherited member
    private:
        double baseSalary;
};
#endif
```

Inheritance – an example

```cpp
// BasePlusCommissionEmployee.cpp
#include <iostream>
#include "BasePlusCommissionEmployee.h"

BasePlusCommissionEmployee::BasePlusCommissionEmployee( const string &first,
    const string &last, const string &ssn, double sales,
    double rate, double salary):

    CommissionEmployee(first, last, ssn, sales, rate)
    {
        setBaseSalary(salary);
    }

double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
}
```

University of Essex

Inheritance – an example

```
// main.cpp
#include <iostream>
#include "BasePlusCommissionEmployee.h"

using namespace std;

int main()
{
    BasePlusCommissionEmployee employee("Bob", "Lewis",
        "333-33-3333", 5000, .04, 300);

    cout << "First name is " << employee.getFirstName()
    << "\nLast name is " << employee.getLastName()
    << "\nSocial Security number is "
    << employee.getSocialSecurityNumber()
    << "\nGross sales is " << employee.getGrossSales()
    << "\nCommission rate is "
    << employee.getCommissionRate() << endl;
    cout << "Base salary is " << employee.getBaseSalary() << endl;
    cout << "Employees earnings: $"
    << employee.earnings() << endl;
}
```

Static and Dynamic Binding

- In the code on the previous slide the **earnings** method from the **BasePlusCommissonEmployee** class will be invoked on the penultimate line since the variable **employee** has that type.
- If we wrote a similar class in Java and then wrote code such as

```
BasePlusCommissionEmployee be = ......;
CommissionEmployee ce = be;
System.out.println(ce.earnings());
```

- the **BasePlusCommissonEmployee** method would be invoked since **ce** refers to an object of the subclass. Java uses ***dynamic binding*** and decides which method to invoke at run time.
- However C++ normally uses ***static binding***; the choice of which method to invoke is made by the compiler according to the type of the variable, not the type of the object.

Static and Dynamic Binding

- The C++ equivalent of the Java code on the previous slide is

```
BasePlusCommissionEmployee be(......);
CommissionEmployee &ce = be;
cout << ce.earnings();
```

- Note the use of a reference variable; if we had written **CommissionEmployee ce = be** we would be making a copy of the inherited part of **be**.
- The above code will invoke the **earnings** member function from the **CommissionEmployee** class since the type of the variable is a reference to this class.
- The fact that the variable refers to an object of the derived class plays no part in the decision as to which function to invoke.

What's not inherited

- The following items are not inherited from a base class:
  - constructors and destructors (the names of these use the name of the class)
  - assignment operators (if the programmer does not provide an assignment operator for a derived class the compiler will generate a default one – this will invoke the assignment operator from the base class)
  - friend functions (they are not members of the class!)

When to use inheritance

- A programmer will often have to make a choice of whether to use inheritance or ***composition*** (i.e. using a class object as a member of another class).
- The general rule is that inheritance should be used for "is-a" relationships, e.g. a student ***is*** a person so it is sensible to write a **Student** class as a subclass of **Person**, but a car ***has*** an engine so a class **Car** should normally have an **Engine** object as one of its members rather than being written as a subclass of **Engine**.

When to use inheritance

- In some circumstances an "is-a" relationship should not be represented using inheritance.
- It could be argued that a stack is a list but it is not appropriate to write a stack class as a subclass of **List** since the latter class has methods that should not be applied to stacks.
- A square is a rectangle but problems may occur if we try to write a class called **Square** as a subclass of **Rectangle**.
- If the latter has methods **setWidth** and **setHeight** a user may change either the width of the height of a square so that it is no longer square.
- The programmer may redefine these methods in the **Square** class so that both will adjust both the width and height, but because of the use of static binding it is not possible to prevent a user from invoking the base class versions.

When to use inheritance

- It would be possible to write **Square** as a subclass of **Rectangle** using protected or private inheritance but this would not allow the user to invoke other methods of the base class, such as **getArea**, and the benefits of inheritance would be lost.
- A wrapper class would probably be more appropriate:

```cpp
class Square
{
    private:
        Rectangle r;
    public:
        Square(int size): r(size, size) {}
        void setSize(int s)
        {
            r.setWidth(s);
            r.setHeight(s);
        }
        int getArea() const
        {
            return r.getArea();
        }
};
```

Virtual Functions

- Consider a class for the storing information about shapes that are to be displayed on some graphical display.
- Each shape will have some attributes such as size, screen position and colour.
- To allow all of the shapes that are to be displayed to be processed uniformly we need to store them in a list or set of objects of the same type; hence we shall need a **Shape** class.
- However, some of the properties of shapes are dependent on the individual shapes: the area of a square is the square of its sides, but the area of a circle is πr^2 .
- Consequently we will need a subclass of the **Shape** class for each type of shape.

University of Essex

Virtual Functions

- Here is an outline of a **Shape** class.

```
class Shape
{
    public:
        Shape(int size, int x, int y);
        void changeSize(int newSize);
        void move(int newX, int newY);
        int getSize(), getX(), getY();
    protected:
      int size, xpos, ypos;
  };
```

- We assume that the shapes being used are squares, circles, equilateral triangles and other regular polygons, so that we do not have to consider the size in terms of width and length.

Virtual Functions

- A **Circle** class can be written as a derived class of **Shape**:

```
class Circle: public Shape
{
    public:
        Circle(int diam, int x, int y): Shape(diam, x, y) { }
        float area() const
        {
            int radius = size/2;
            return M_PI * radius * radius;
        }
};
```

- Note that the header file **<cmath>** needs to be included in order to use **M_PI**.

Virtual Functions

- We can write similar classes **Square** and **Triangle**; although the areas of the squares will be integers, all of the **area** methods should return results of type **float** so that all shape classes have similar functionality.
- To calculate the total area of all of the shapes in a collection **c** of pointers to objects of type **Shape** (assuming that all of the objects belong to derived classes that have **area** methods) we would wish to be able to write a loop of the form

```
float totalArea = 0.0;
for (it = c.begin(); it != c.end(); it++)
    totalArea += (*it)->area();
```

Virtual Functions

- The code on the previous slide will not compile since the **Shape** class does not have an **area** method so the compiler will not accept **(*it)->area()**.
- We could write a version that tries out several different dynamic casts (to be covered in part 10) but this would be cumbersome and we would have to know the names of all of the subclasses so the code would have to be modified if new subclasses were created.
- In Java we could simply give the **Shape** class an **area** method that returns 0.0;
- due to dynamic binding the appropriate subclass method would get invoked for each object in the collection.
- This would not work in C++ since static binding results in the method from the **Shape** class being invoked for each object.

Virtual Functions

- To get dynamic binding in C++ we have to declare a member function in a base class to be a ***virtual function***.
- This is done by preceding its name with the keyword **virtual**.
- Hence we should add to the public part of the **Shape** class the function definition

  virtual float area() const { return 0.0; }
- The function in the derived class should not be declared as virtual unless we expect to further extend this class with other subclasses that will need different versions of the function, so we should not change the declaration of the **area** function in the **Circle** class.

Virtual Functions

- When a function declared in a base class as virtual is applied to an object of a derived class accessed using a pointer or reference to the base class, the derived class version of that function overrides the inherited version and will be invoked.
- Note that the dynamic behaviour of virtual functions is only achieved when pointers or references are used since, for example, a variable of type **Shape** cannot hold a **Circle** object.
- Also note that if a derived class has a function with the same name as a virtual function of the base class, but with different argument types, the derived-class version will not override the virtual function.

Virtual Functions

- Consider the following code.

```
Circle c(12, 4, 8);
Shape s1 = c;
Shape &s2 = c;
Shape *p = &c;
cout << s1.area() << endl;
cout << s2.area() << ',' << p->area() << endl;
```

- In the first assignment only the inherited part of **c** is copied into **s1** so **s1** is not a circle, and the base class **area** function will be invoked in the first output statement.
- The variable **s2** refers to a circle and the pointer **p** points to a circle, so since **area** is a virtual function the derived class version will be invoked twice in the second output statement.

Abstract Classes

- An ***abstract class*** is one that is used purely as a base class; no instances of it are allowed that are not instances of derived classes.
- In C++ a different technique is used: a class is abstract if it has a ***pure virtual function***.
- This is a function that has no implementation in the base class and is declared using the syntax

virtual float area() const = 0;

- As in Java all concrete subclasses of an abstract class must provide versions of the function to override the pure virtual version.

Abstract Classes

- Here is an abstract version of the **Shape** class.

```
class Shape
{
    public:
        Shape(int size, int x, int y);
        void changeSize(int newSize);
        void move(int newX, int newY);
        int getSize(), getX(), getY();
        virtual float area() const = 0;
    protected:
      int size, xpos, ypos;
};
```

Abstract Classes

- Since no instances of an abstract class that are not instances of derived classes can be created it is not possible to have variables whose type is the abstract class;
- we must use references and/or pointers, so a declaration such as **Shape s;** would not be allowed. The following would, however, be permissible:

  Shape &s = Circle(6, 10, 10)

  Shape *p = new Square(5, 20, 20);
- A declaration such as **Shape s[10];** is also not allowed and the type of objects in an STL collection cannot be an abstract class.

University of Essex

Abstract Classes

- Pointers to abstract classes may be used as template arguments for the STL containers so it is possible to declare collections of **Shape** objects via pointers:

```
Vector<Shape*> v;
v.push_back(new Circle(10, 20, 20));
v.push_back(new Triangle(5, 30, 30));
..........
for (int i = 0; i < v.size(); i++)
    cout << v[i]->area() << endl;
```

University of Essex

Abstract classes and output

- Suppose we want to use **cout << x**, where **x** is a reference variable of type **Shape&**.
- Since **x** must refer to an object of a class derived from **Shape**, we would probably want the output to depend on the class of this object.
- However, since **operator<<** cannot be written as a member function of **Shape** we cannot make it virtual.
- Instead we need to write an **operator<<** function that calls a virtual function to perform the actual output:

```
ostream &operator<<(ostream &o, const Shape &s)
{
    s.put(o);
    return o;
}
```

Abstract classes and output

- In the class **Shape** the function **put** would be declared as apure virtual function:

virtual void put(ostream&) const = 0;

- Each subclass would contain a version of **put** that outputs the contents of that class to the stream.
- For example in the **Square** class we might use something like

```
void put(ostream &o) const
{
    o << "Square of size " << size << " at ("
    << x, << ',' << y << ')';
}
```

University of Essex

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex