CE221

Programming in C++
Part 8
Inheritance, Exceptions
Inheritance 1

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 private members of the base class cannot be accessed directly in the methods or friend functions of the derived class; if we need to access these we must do so by calls to methods or friend functions of the base class.
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, e.g. if the base class has a public member `age` and `s` is an object of the derived class we can access `s.age` anywhere in the program (as long as the classes and object are in scope).

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.
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, e.g. a derived class may have an `operator string` function that is different from that of its base class.
Inheritance 4

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 5

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

```cpp
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 6

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 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.
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.

(Note that this does not apply the other way round; we cannot use `Student s(p)` or `s = p`.)
Inheritance 8

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 ::.

The next slide presents a possible print method for the Student class which uses the inherited print method to print the name.
Inheritance 9

```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 1

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.

(To save space the example is incomplete – the definitions of some member functions that are declared in the header files have been omitted from the `.cpp` files. Writing them would be easy.)
/ CommissionEmployee.h
#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;
    // class continued on next slide
Inheritance – an Example 3

// CommissionEmployee.h continued
// class CommissionEmployee continued
// public members continued
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
/ CommissionEmployee.cpp
#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 5

// 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 6

// 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();
}

// other member function definitions needed
// (set/get functions, print)
/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 << Employee's earnings: $" << employee.earnings() << endl;
}
Static and Dynamic Binding 1

In the code on the previous slide the `earnings` method from the `BasePlusCommissionEmployee` 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

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

the `BasePlusCommissionEmployee` 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.
The C++ equivalent of the Java code on the previous slide is

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

Note the use of a reference variable; if we had written

```cpp
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 1

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 2

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.
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) {}  // Constructor
        void setSize(int s)
            { r.setWidth(s); r.setHeight(s); }  // Method to set size
        int getArea() const { return r.getArea(); }  // Method to get area
};
```
Exceptions 1

It is often the case that the handling of an error needs to be done in a different part of the program than the place where it is detected. For example if something goes wrong when a function is called the caller may want to display an appropriate message.

The detector of the error needs to indicate in some way that an error has occurred. This could be done using a global flag (e.g. `if (x<0) error = true;`) or by returning a special value (e.g. `if (x<0) return -99999;`), but the preferred approach is to throw an exception (e.g. `if (x<0) throw tantrum();`)
Exceptions 2

C++'s approach to exception handling uses `try` and `catch` blocks in the same way as Java. An exception should normally be an object whose type is either `exception` (as defined in the header file `<stdexcept>`) or some subclass of `exception`, but (unlike Java) it is permissible to use a class that does not inherit from `exception`.

The `exception` class has a public member function called `what` that will return a C-string containing a description of the exception.

The header file `<stdexcept>` defines several exception classes that may be thrown by standard library functions; the hierarchy shown on the next slide shows most of these classes.
Exceptions 3

exception

logic_error
invalid_argument
length_error
domain_error
out_of_range

runtime_error
overflow_error
underflow_error
range_error

bad_alloc
bad_cast
bad_type_id
bad_exception
Exceptions 4

The two subclasses of the `exception` class, `runtime_error` and `logic_error`, have constructors that take a constant reference to a string as an argument; the inherited `what` function will return, as a C-string, the string which was supplied when the object was created.

When a user wishes to create his own exception classes the usual practice is to use one of these as the base class.

The former should be used when the error is of a nature that could not be prevented by the programmer (e.g. failure to open a file) whereas the former should be used if the programmer could have prevented it (e.g. index out of range).
If we wish to write a named exception class that behaves in the same way as one of these classes we would simply use something like

```cpp
class MyException: public runtime_error
{
public:
    MyException(const string &s): runtime_exception(s) {}
}
```

This allows users to write `catch` blocks that just catch occurrences of `MyException`, for example.

```cpp
try{
    ........
}

catch (MyException e)
{
    cout << "Caught exception: " << e.what();
}
```
In many applications we may wish to generate message strings containing some fixed text and additional text supplied as an argument to the exception constructor.

```cpp
class StudentException: public logic_error
{
public:
    StudentException(const string& s):
        logic_error("Student problem: "+s)
    {
    }
}
```
If we need to generate a message string that cannot be created in an expression that can be used in an initialiser list we should instead store the string as a private data member and redefine the `what` function to return it.

```cpp
class StudentException: public exception
{
  private char mess[100];
  public:
    StudentException(const string &s):
    { // generate message in mess
    }
    const char *what()
    { return mess;
    }
}
```

Exceptions 6

The use of `try`, `catch` and `throw` is the same as in Java:

```cpp
int myFunc(int n)
{
    if (n<0) throw MyException("n is negative");
    .......
}

....... try
{  x = myFunc(y);
    .......
}
catch(MyException e)
{  cout << "Invalid argument to myfunc" << endl;
}
```
As in Java we can specify which classes of exceptions could potentially be thrown by a function. C++, however, has no `throws` keyword, so `throw` is used with the following syntax:

```cpp
int myFunc(int n) throw (MyException, HisEx)
{
    if (n==0) throw MyException("n is zero");
    if (n<0) throw HisEx(n);
    ....
}
```

Observe that parentheses are always used, even if there is only one exception class in the list.

Unlike Java there is no concept of checked and unchecked exceptions so the choice of whether to use an exception-list in a declaration is left to the programmer.
Exceptions 8

We can use `throw()` in the declaration to indicate that the function will not throw an exception. This does not however guarantee that an exception will not be thrown since the compiler is unable to check whether any calls made inside the function body to functions written in a separate file can potentially throw exceptions if their declarations do not contain a throw-clause.

If a function does try to propagate an exception that was thrown by a called function but is not listed in its throw-clause an exception of type `bad_exception` will be thrown instead.
Exceptions 9

As in Java we can use `catch(exception e)` to catch all exceptions that derive from `exception`; this will not however catch any exceptions that are not objects derived from this class. A catch block headed `catch(...)` will catch anything that is thrown, but since this may be an object of any type we cannot examine it within the block. (The `...` is part of the syntax, i.e. exactly three dots without any spaces.)