Inheritance

Inheritance - Review

- Inheritance is an important feature of the Object Oriented paradigm
- Allows a new class to be defined in terms of an existing class
  - Additional functionality can be added
  - Functionality of the existing class can be changed
- Existing class called the base class
- New class called the derived class
- C++ supports conventional inheritance model

Example

- A "plumbing job" is like a normal job, but there is a fixed callout charge that has to be applied

```
class PlumbJob : public Job
{
private:
    int  calloutCharge;
public:
    void SetCallout( int calloutValue ) { calloutCharge = calloutValue; }
};
```

Name of base class. The 'public' means that public members of the base class become public members of the derived class. 'private' (the default) means that public members of the base class become private members of the derived class, and is rarely used

PlumbJob inherits all the functionality of Job but in addition has the calloutCharge member and the SetCallout function to sets its value

Using the Derived Class

- The object of the derived class can call member functions defined within itself
  - And (non-private) member functions defined in the base class

```
#include "job.h"
#include "plumbjob.h "
...
PlumbJob pj;
pj.SetHours( 2 );
pj.SetRate( 30 );
pj.SetJobNum( 1234 );
pj.SetCallout( 50 );
int cost = pj.GetAmount();
```

Inherited functions

Defined in PlumbJob

The inherited GetAmount function can be called, but it gives the wrong answer as it doesn't include the callout charge

Overriding

- The derived class can define a local version of a function defined in the base class
  - To account for features specific to the derived class

Class definition, PlumbJob.h:

```
class PlumbJob : public Job
{   ...
public:
    void SetCallout( int calloutValue ) { calloutCharge = calloutValue; }
    int GetAmount();
};
```

Overrides the base class member function with the same name and argument types

Implementation, PlumbJob.cpp:

```
int PlumbJob::GetAmount()
{
    return Job::GetAmount() + calloutCharge;
}
```

Calls the base class function to calculate the basic job cost by preceding it with the base class name

Inheritance, Constructors and Destructors

- Creating instance of derived class requires creation of instance(es) of base class(es)
  - Constructors invoked in order from base to derived
- Destruction of derived object requires destructor to be called on instance(s) of base class(es)
  - In reverse order of constructor invocation
- Compiler automatically calls default (void) constructor(s)
  - Problem if no void constructor defined on base class
  - Or if a specific constructor is to be called to process arguments

Example

- Job and PlumbJob

```
class Job
{
public:
    Job( int hoursVal, int rateVal, int jobNumVal );
    ...
};
class PlumbJob : public Job
{
public:
    PlumbJob( int hoursVal, int rateVal, int jobNumVal, int calloutVal );
    ...
};
```

Base class constructor would typically copy the arguments into its private members

Derived class constructor also has arguments to use for the base class members

Base Class Constructor Invocation

- Uses Initialiser List
  - Ensures appropriate base class constructor is called before local constructor
  - Can also be used in non derived class constructor to perform efficient initialisation of members

```
PlumbJob::PlumbJob( int hoursVal, int rateVal, int jobNumVal, int calloutVal )
  : Job( hoursVal, rateVal, jobNumVal )
{
    calloutCharge = calloutVal;
}
```

The base initializer comes before the first '{', emphasizing that it is used before the body of this function.

Note the initial ':' and the absence of a trailing ';'

Member Initialisation

- Initialiser list can also be used to initialise members of local class
- *Must* be used if
  - Member is const
  - Member is a reference
  - Member is an object requiring a specific non-void constructor to be called
- Intialiser list has additional advantages
  - Cannot intialise a member twice
  - Initialisation is performed in order of declaration of members in the class, not order in initialiser list, making it easier to handle dependencies between members

Example

- Assume the presence of a class Worker, and a slightly modified PlumJob class
  - Worker object is constructed using name and number

```
class PlumbJob : public Job
{
private:
        const int calloutCharge;
        Worker worker;
public:
        PlumbJob( int hoursVal, int rateVal, int jobNumVal, int calloutVal );
        …
```

Constructor Using Initialiser List

```
PlumbJob::PlumbJob(  int hoursVal, int rateVal,
                     int jobNumVal, int calloutVal )
  : Job( hoursVal, rateVal, jobNumVal ),
    calloutCharge( calloutVal ),
    worker( "fred", 6 )
{
}
```

The base class constructor is called as before

The const callout charge is given a value

Initializers are separated by a ‘,’

The constructor arguments for the CWorker object are given

Using Pointers

- Declare 2 Job * pointers
  - Inheritance means the pointers may point to a Job or a PlumbJob

```
Job * pJ1;
Job * pJ2;
int cost;

pJ1 = new Job;
cost = pJ1->GetAmount();

pJ2 = new PlumbJob;
cost = pJ2->GetAmount();
```

Although pJ1 and pJ2 are declared as type Job *, this really means that they can point to any object which is at least a Job

We can use pJ2 to point to a PlumbJob as a PlumbJob is ‘at least a Job’

GetAmount gives the wrong answer. The compiler generates a call to Job’s GetAmount, as this is the pointer type

Virtual Functions and Polymorphism

- If a base class pointer is initialised to point to an instance of the derived class, calling a member of the base class will call the base class function
  - Even if the derived class overrides
  - Static binding
- Defining a function as *virtual* causes the overridden function to be called
  - Dynamic binding

```
class Job
{
        ...
        virtual int GetAmount();
};
```

The virtual keyword is used in the base class Job, which first declares the GetAmount function

If a class contains at least one virtual function, the destructor should also be virtual

The override Keyword

- Marks a method as available for overriding
  - Added in C++11
  - Earlier versions used method signature to implement overriding

```
class PlumbJob : public Job
{
public:
        int GetAmount() override;
        ...
};
```

```
class PlumbJob : public Job
{
public:
        int GetAmont() override;
        ...
};
```

```
'PlumbJob::GetAmont' : method with
override specifier 'override' did not
override any base class methods
```

Polymorphism Through Inheritance

- "Inclusion Polymorphism"
- Pointer to base class type can be used to point to a derived class object
  - Most derived version of any virtual function will be called
- Uses *vtable* to implement this
  - Table of function pointers
  - Every object with at least one virtual function will have a vtable
  - Virtual function calls are made indirectly through the vtable
  - Introduces a slight overhead

```
Job * pJ2;

...

pJ2 = new PlumbJob;
cost = pJ2->GetAmount();
```

The correct implementation of GetAmount is used, based on the type of object pointed to

Polymorphism in Action

- Common use cases associated with collections

```
class Jobs
{
private:
      Job * jobList[10];
      int count;

public:
      Jobs();
      void AddJob();
      void AddPlumbJob();
};
```

```
Jobs::Jobs()
{
      count = 0;
}

void Jobs::AddJob()
{
      jobList[count++] = new Job;
}

void Jobs::AddPlumbJob()
{
      jobList[count++] = new PlumbJob;
}
```

Either Job or PlumbJob objects are added to the array depending on the function used

Inheritance and Encapsulation

```
class Job
{
private:
      int  hours;
      int  rate; ...
public:
      ...
      int  GetAmount();
};
```

```
class PlumbJob : public Job
{
private:
      int  calloutCharge;
public:
      ...
      int  GetAmount();
};
```

```
int Job::GetAmount()
{
      return rate * hours;
}
```

We call the existing method in the base class to perform the basic cost calculation, then add on the extra callout charge

```
int PlumbJob::GetAmount()
{
      return Job::GetAmount()
            + calloutCharge;
}
```

Inheritance and Encapsulation

- Alternatively

```
int PlumbJob::GetAmount()
{
      return hours * rate + calloutCharge;
}
```

This looks ok at first glance, but is a bad idea because:

1. We are reproducing functionality which has already been written, tried and tested
2. hours and rate are private in the base class, so can only be accessed by methods in that class, even though PlumbJob inherits them

Protected Members

- The protected keyword introduces a further option for visibility control
  - Visible in current class and any derived class

```
class Job
{
protected:
      int  hours;
      int  rate; ...
public:
      ...
      int  GetAmount();
};
```

```
int PlumbJob::GetAmount()
{
      return hours * rate
          + calloutCharge;
}
```

This is still a bad idea though, as we are rewriting existing functionality, perhaps introducing new errors

Using Protected Member Functions

- Provide accessor methods for private data in base class to derived classes only
  - Allows "read only" access by the derived classes

```
class Job
{
private:
          int  hours;
          int  rate; ...
protected:
          int GetHours() { return hours; }
          int GetRate() { return rate; }
          ...
};
```

Extending the Model

- Introduce FixedPriceJob, with cost not based on hourly rate
  - Ideally needs to be derived from Job
  - Doesn't use hours or rate members, since price is fixed
- Current Job class is to specific
- Alter implementation of base class to manage requirement
  - GetAmount() function should deal with above cases
  - We define an *abstract class*

An abstract class cannot be used to create objects (it cannot be 'instantiated'). It contains at least one unimplemented 'pure virtual' function which must be overridden in classes which derive from it (unless the derived class is also abstract)

Defining the Abstract Class

- Abstract class defines properties that are shared by all derived classes
  - Not necessary to provide implementations
  - Use *pure virtual functions*

```
class AbstractJob
{
private:
    int jobNum;
public:
    void SetJobNum( int jobNumValue ) { jobNum = jobNumValue; }
    virtual int GetAmount() = 0;
};
```

There is no specific keyword to make a class abstract in C++, a class is abstract if it contains at least one pure virtual function

Pure virtual functions begin with virtual and end with '= 0'. They do not have a body

The Job Class

- Job class now derives from AbstractJob
  - PlumbJob does not change

```
class Job : public AbstractJob
{
private:
        int   hours;
        int   rate;
public:
        int  GetAmount();
        …
};
```

Job implements the GetAmount function which was pure virtual in AbstractJob. It is therefore a 'concrete class' which we can instantiate

FixedPriceJob

- FixedPriceJob also derives from AbstractJob
  - Provides its own implementation of GetAmount()

```
class FixedPriceJob : public AbstractJob
{
private:
        int cost;
public:
        void SetCost( int costValue )
        {
                cost = costValue;
        }
        int GetAmount() { return cost; }
};
```

The Jobs Collection Class

- Alter this to hold an array of AbstractJob pointers
  - These can only point to concrete instances of the derived classes

```
class Jobs
{
private:
        AbstractJob * jobList[10];
        int count;
public:
        Jobs();
        void AddJob();
        void AddPlumbJob();
        void AddFixedPriceJob();
};
```

dynamic_cast

- Used to convert a pointer to a base type into a pointer to a derived type
  - For example to give access to members specific to the derived type
- Returns null pointer if cast cannot be performed
  - I.e. pointer does not refer to the expected derived type

```
FixedPriceJob * pFPJ;
pFPJ = dynamic_cast<FixedPriceJob *>( jobList[i] );
if ( pFPJ )
{
        pFPJ->SetCost( 123 );
}
```

jobList is an array of AbstractJob pointers

The dynamic_cast will only succeed if the pointer is to a FixedPriceJob or an object of a class derived from it. Otherwise it will return 0

Only calls the function defined by FixedPriceJob if the dynamic_cast returns a non-null pointer

dynamic_cast requires RTTI

RTTI: Run Time Type Information

- Small amount of metadata available at runtime to support dynamic cast operations
- typeid() function returns object of type type_info, containing (a small amount of) information about a class
  - Enough to support == and != operators
  - Equality should only be true if types match exactly
  - Matching against subtype may return true, depending on compiler

```
if ( typeid( *jobList[i] ) == typeid( Job ) )
{       ...     // It is a Job
```

Note the * used as jobList is an array of pointers, otherwise typeid will use the pointer type

typeid can be used for a class or an object

Type Name

- Returned by name() function
  - Returned as char * referring to a statically allocated string
  - Must not be deleted

```
string str;

cout << typeid( str ).name();
```

gives:

```
class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >
```