

Inheritance

1. Introduction

One of the most important concepts in OOP is 'reuse', the ability to reuse code in many different applications. This is a feature that prevails heavily in 'Windows' applications. For example, a dialog box, this is used again and again but has many different contents or commands. This reuse of code can be achieved by client - server implementation, where existing classes are incorporated into a program but depends on the program being structured into discrete objects with well defined interfaces. Alternatively a technique called inheritance can be used. Inheritance takes existing classes, modifies them and incorporates the modified class into the program. The availability of inheritance depends upon the language supporting openness.

Inheritance allows objects to acquire the attributes and behaviour of other objects. This contributes to economical and maintainable design, because objects share attributes and behaviour without separately duplicating the program code that implements the objects.

A good analogy is the taxonomic scheme used by zoologists and botanists to classify living things. This method of classification divides the plant and animal kingdom into a number of groups. Each group is in turn subdivided into classes, orders, families, and so on. Lower-level groups inherit, or share the characteristics of higher-level groups. For example a wolf is a member of the canine family or class which is a subgroup of carnivores which have the characteristics of being meat eating, this in turn is a sub group of mammal, which have the characteristics of warm

blood and hair, this in turn is a subgroup of the vertebrates that is a group that has backbones, finally vertebrates are a subgroup of animals. It can be seen from this example that the wolf inherits characteristics from the super or parent groups or classes in the hierarchy. This hierarchy relationship is shown in figure 1.

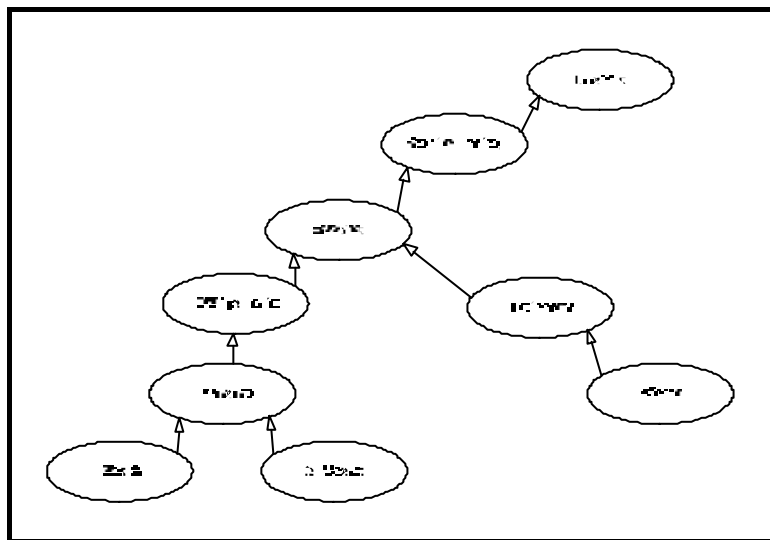


Figure 1: Inheritance

Software objects or classes can occupy a hierarchy in much the same way. It is this hierarchy that leads to inheritance by a process which allows class definitions to be reused and modified. The original class is known as the superclass (or parent, or base class) and the new derived class is known as the subclass (or child or derived class). A subclass can have added to it new behaviour, ie data members and member functions, to that of the

superclass.

2. Specialisation

This leads to a facility known as specialisation, that is a subclass is a specialisation of a superclass. In other words, our class canine deals with special occurrences of the class carnivore. Conversely, a superclass is a generalisation of a subclass, or mammals generally explain that group or class of animals.

To establish the hierarchy of an inherited class and its relationship we use the IS-A rule. In the above example a wolf IS-A a canine, a canine IS-A carnivore, a carnivore IS-A mammal, a mammal IS-A vertebrate, and a vertebrate IS-A animal.

3. Inheritance Syntax

The C++ syntax for creating an inherited class requires that the superclass is also included in the declaration, so first we must declare the superclass.

```
//superclass
class Animal{
    public:
        .....the interface....
    private:
        .....the state.....
};
```

The subclass declared here is Mammal and included in its declaration is its superclass Animal:

```
//subclass
class Mammal : public Animal{
    public:
        .....the interface.....
    private:
        .....the state.....
};
```

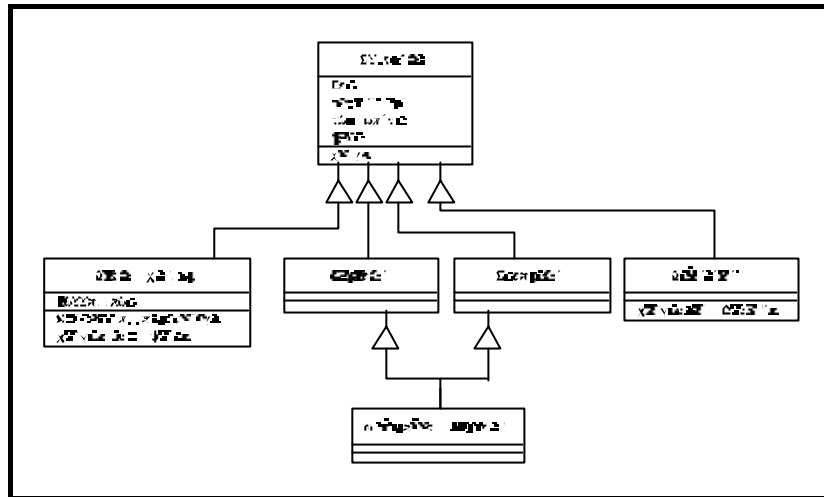
Mammal is a subclass derived from the superclass Animal.

The use of the access specifier public, in the declaration of the subclass of mammal means that all public members of the superclass (animal) are inherited as public members of the subclass, and all protected members of the superclass are inherited as protected members of the subclass. If the access specifier 'private' were used, it would mean that all public and protected members of the superclass are inherited as private members of the subclass.

The private members of the superclass are not visible to the subclass and should be declared as protected in the superclass if it is intended that the subclass should have access.

4. Implementing Inheritance

There are two methods of implementing inheritance, inheritance by extension and inheritance by overloading. Lets consider a model of a company:



Inheritance by Extension

When a subclass is created it inherits all the member functions (the interface) and data members (the state) from that of the superclass. In the inherited class (the subclass) additional member functions that are only required for the subclass can be added. This extension allows the specialisation of the subclass by these additional member functions (or interface) being only applicable to the subclass. Inherited classes (subclasses) can also have in their state additional data members, i.e. attributes that only applies to that subclass. For example, if we had a superclass Employee, which may have an attribute JOB TITLE, we may have two subclasses Sales Person and Secretary. The Sales Person subclass may have an additional attribute of COMMISSION, and an additional interface of GET_COMMISSION_RATE(). Both of these would specialise the subclass Sales Person to hold additional data relevant only to a Sales Person.

```
\\superclass
class Employee{
    public:
        Pay();
    private:
        int Age;
        char *JobTitle
        char *Name
        char *Department
};

//subclass
class SalesPerson : public Employee{
    public:
        //subclass inherits Pay();
        Get_Commission_Rate();//function added in subclass
    private:
        //subclass inherits Age, JobTitle, Name & Department
```

```

        float Commission;           //data member added in subclass
    };

```

Inheritance by Overloading

We have already encountered function overloading in earlier sessions. The principle is the same when creating subclasses. A function that exists in the superclass can be overloaded in a subclass. For example in our superclass of Employee, we may have a function for PAY(), in our subclass of Sales Person, the function PAY() may be overloaded to select the commission rate, PAY(double Rate), and in our subclass Secretary the function would identify the annual salary, PAY(float Salary).

```

\\superclass
class Employee{
    public:
        Pay();
    private:
        int Age;
        char *JobTitle
        char *Name
        char *Department
};

//subclass
class SalesPerson : public Employee{
    public:
        //subclass inherits Pay();
        Pay(double Rate); //function Pay overloaded
        Get_Commission_Rate();//function added in subclass
    private:
        //subclass inherits Age, JobTitle, Name & Department
        float Commission; //data member added in subclass
};

```

5. Multiple Inheritance

Subclasses can inherit the interface and state from more than one superclass. For example if we were to extend our model of Employee >Manager, Engineer and include Technical Manager, Technical Manager IS-A Manager and Technical Manager IS-A Engineer, thus giving multiple inheritance for Technical Manager from Manager and Engineer.

6. Structure of Source Code and Project Files

The classes you develop, you may want to use in a number of different programs. This leads to a conventional structure for the source code files of a C++ program. By saving the file sever times under appropriate names and deleting the unwanted parts, make the following files:

a header file called by the class name e.g. `Animal.h`, this contains just the class declaration;

a file called in this example `Animal.cpp` containing just the definitions of the

member functions for the class but no class declaration. You must add `#include "animal.h"` at the top of this file.

The reason for structuring the source in this way is:

we can separately compile the `animal.cpp` and either keep a separate object file `animal.obj` or store the object file in an object library for later use;

we can also separately compile the `main` function (in our client program e.g. `zoology.cpp`), which knows about the *animal* class interface because it has included the `animal.h` file. Each time you change the `main` function, you do not need to recompile the *animal* class, simply relink the new `main` with the `animal` object file.

In order that the compiler can know where to look for the separate files, you need to use a *project* to manage your program. On the 'project' menu, select 'open project...', and enter the name for your project, e.g. `zoology.prj` (the `prj` extension will be given automatically). This will create a new project file called `zoology.prj`, and a project window will appear at the bottom of the screen. With the project window highlighted, select 'add item..' from the project menu, then add the files `zoology.cpp` and `animal.cpp` to the project. Do not add the header files e.g. `animal.h` to the project, but do make sure it is in the same directory as the files `zoology.cpp` and `animal.cpp`.

Now with the project open, if you select 'make' from the 'compile' menu then the separate files in the project will be compiled and linked together. If you end your session and restart later, you do not need to create a new project file, since the file `zoology.prj` has been saved on your disk. Simply open the project file from the 'project' menu, and carry on working. You should always try to organise your source code in this way, keeping a separate project file for each different system you are working on.

CREATING A PROJECT FILE

```
//File Name: myprog.cpp
#include <iostream.h>
#include <conio.h>
#include "a:classa.h"
#include "a:classb.h"

void main(void)
{
    ClassA obj1;
    ClassB obj2;

    clrscr();
    obj1.get_num1();
    obj2.get_num2();
    getch();
}

//===== end of program =====

//=====
//File Name: classa.h
//Header file for classa.cpp & myprog.cpp
class ClassA
{
    private:
        int num1;
    public:
        void get_num(void);
};
//=====
//File Name: classa.cpp
//member functions for class ClassA
#include <iostream.h>
#include "a:classa.h"

void ClassA::get_num1(void)
{
    cout << "Enter first Numer ";
    cin >> num1;
    cout << "\nThe number entered was " <<
num1;
}
//=====

//=====
//File Name: classb.h
//Header file for classb.cpp & myprog.cpp
class ClassB
{
    private:
        int num2;
    public:
        void get_num(void);
};
//=====
//File Name: classb.cpp
//member functions for class ClassB
#include <iostream.h>
#include "a:classb.h"

void ClassB::get_num2(void)
{
    cout << "Enter second Numer ";
    cin >> num2;
    cout << "\nThe number entered was " <<
num2;
}
//=====
```