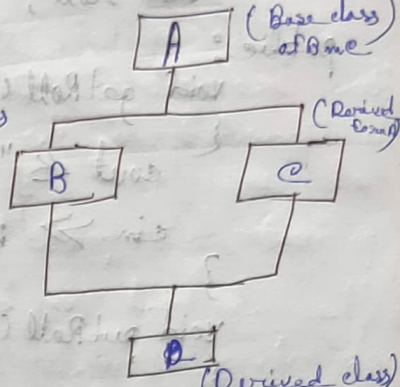
Hybrid Inheritance:

A class (child class) is derived from two or more base class and those base class are derived from a base class. It known as hybrid inheritance.

- It is a combination of multi-level, multiple, and hierarchical inheritances.

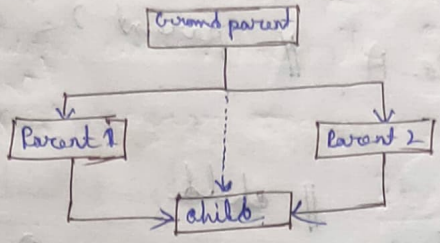


Derive class with hybrid inheritance is declared by following:

```
                    // base class
class A
{
  ____
  ____
};
class B : public A    // B derived from baseclass A
{
  ____
  ____
};
class C : public A    // C derived from base class A
{
  ____
  ____
};
class D : public B, public C    // D derived from base class B and C both
{
  ____
  ____
};
```

Virtual Base class:

A virtual base class ensures that only one copy of the public and protected members are inherited into grand-child ~~class~~ class in hybrid inheritance.



Here, the 'child' class has two direct base classes 'parent 1' and 'parent 2' which themselves have a common base class 'grandparent'. All the public and protected members of 'grandparent' are inherited into 'child' twice first via 'parent 1' and again via 'parent 2'. This means, 'child' class would have duplicate sets of members inherited from 'grandparent'. This introduces ambiguity and should be avoided. This can be avoided by making the common base class as virtual class.

Example:

see — the book. 204

```
class GrandParent
{
    // member of base class
};
class Parent1 : virtual public GrandParent
{
    // meber of child class
};
class Parent2 : virtual public GrandParent
{
    .
    .
};
```

12018

Supper class →

The class whose properties are inherited by sub class is called Base class / super class

Inheritance by the 'child' as shown in Fig. 8.12 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have *duplicate* sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

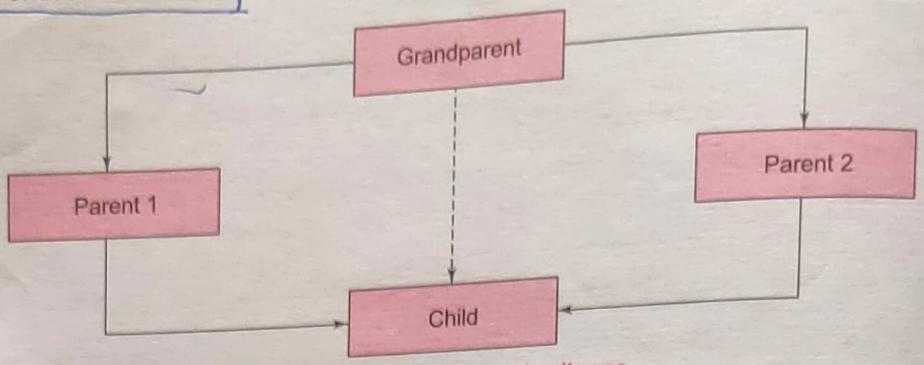


Fig. 8.12 *Multipath inheritance*

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as *virtual base class* while declaring the direct or intermediate base classes as shown below:

```
class A                              // grandparent
{
    .....
    .....
};
class B1 : virtual public A          // parent1
{
    .....
    .....
};
class B2 : public virtual A          // parent2
{
    .....
    .....
};
class C : public B1, public B2       // child
{
    .....                            // only one copy of A
    .....                            // will be inherited
};
```

When a class is made a **virtual** base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

Note *The keywords* ***virtual*** *and* ***public*** *may be used in either order.*

Abstract class:

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other class). (It is a design concept in program development and provides a base upon which other classes may be built.) — not necessary to read

Abstract class contains atleast one Pure virtual function in it. A pure virtual function is specified by placing "=0" in its declaration.

Example

```
#include <iostream>
using namespace std;
class Shape
{   virtual void area() = 0;
};
class Rectangle : public Shape
{
   void area()
   {
      cout << "Inside the Rectangle" << endl;
   }
};
class Circle : public Shape
{
   void area()
   {
      cout << "Inside the Circle" << endl;
   }
};
int main()
{
   Shape * S;
   Rectangle R;
   S = &R;
   S -> area();
```

Virtual Function

When we use the same function name in both the base class and derived class, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.

Run time polymorphism is achieved only when a virtual function is access through a pointer to the base class. It cannot be achieved using object name along with the dot (.) operation to access virtual function.

We can have virtual destructor but not virtual constructor.

- It can not be static members.
- They are accessed by using object pointer.

Example - Make

(Contd.)

Pure virtual function :

A virtual function, equated to zero is called a pure virtual function.

A class containing such pure virtual function, is called an abstract class.

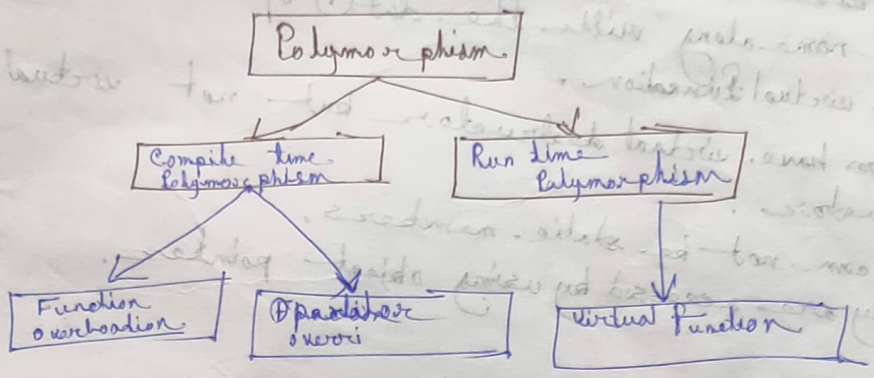
To mention that the abstract class has been defined as "do-nothing" function. Don't need to write any function.

Polymorphism

Polymorphism :

Polymorphism means 'one name having multiple forms'.
~~Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.~~

In C++ polymorphism is mainly classified into two categories as shown below :



Compile time polymorphism	Runtime polymorphism
~~I) In this method~~	I)
I) In this method, object is bound to the function call at the compile time itself.	I) In this method, object is bound to the function call at the run time.
II) The overloaded member function are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and therefore, compiler is able to select the appropriate function for a particular call at the ~~compile time itself~~.	II) The function is linked with a particular class much later the compilation, is known as dynamic/runtime polymorphism, because the selection of the appropriate function is done dynamically at run time.
III) Compile time polymorphism is achieved in two ways: function overloading and operator overloading	III) Virtual function are used in run time polymorphism
IV) It is also known/called early binding/static binding/static linking	IV) ~~—~~ late binding/dynamic binding

Friend Function

```
#include <iostream.h>
using namespace std;
class Sample
{
  private:
    int i;
  public:
    friend void set (Sample);  // friend function is declared not defined within the class
};
void set (Sample A)  // The friend function is defined here the outside of the class
{
  A.i = 10;
}
int main ()
{
  Sample X;
  set (X);  // cannot be called using the object of the class.
  return 0;
}
```

Definition:

A friend function is used to access the private member of a class. The friend function is declared with the keyword friend.

Characteristics:

i) Friend function is declared not defined within the class.

ii) Friends " " defined the outside of class.

iii) It can be invoked like a normal function without the help of any object.

iv) It can be declared either the public or private part of a class without effecting its meaning.

v) It has the objects as argument.

Containership or nesting :

When a class contains objects of another class as its members, this kind of relationship is known as containership.

- Containership represents a "has-a" relationship.
- The class which contains objects of another class as its members is called as container class.

Inheritance	Containership
i) When a class is deriving from another class, this is known as inheritance	i) When a class contains the objects of another class as its members, this kind of relationship is known as containership.
ii) Inheritance represents a "is-a" relationship.	ii) Containership represents a "has-a" relationship
iii) The IS-A relationship is inheritance	iii) The has-a relationship is composition

is-a

has-a

Example

```
class alpha { ... };
class beta { };

class gamma
{
    alpha a;    // a is an object of alpha class
    beta b;     // b is an object of beta class
    ------
};
```

Template function :

A function generated from template is called a template function.

(six) (2015)

Function overloading	Function template
i) Separate implementation is required for each different data type	i) Only one implementation is required for all data type
ii) Function overloading is used when multiple function do similar operation.	ii) Templates are used when multiple function do identical operation

Ex.

```
int add (int a, int b)
{ return a+b
}
float add (float c, float d)
{ return c+d
}
int main()
{
add (2, 3);
add (2.5, 2.6);
```

Ex.

```
T add (T a, T b)
{ return a+b
}

int main()
{
add (2, 3)
add (2.5, 2.6)
}
```

(Contd.)

Exception Handling

What is exception?

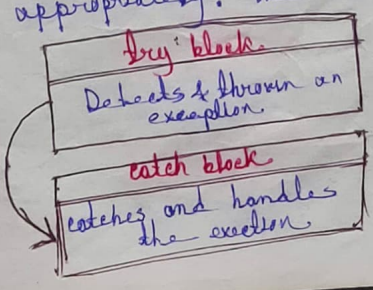
- Exception is a peculiar problem that a program encounter at run time.
- A c++ exception is response to exceptional circumstance that arise while a program is running such as an attempt to divide by zero.
- Exception provide a way to transfer control from one part of a program to another, called handler.

Exception handling Mechanism:

- Exception handling is a process to handle run time errors of a program.
- Exception handling consists of following parts:
  1. Find the problem (Hit the exception)
  2. Inform that the error has occurred (throw the exception)
  3. Receive the error information (catch the exception)
  4. Take corrective actions (handle exception)
- The error handling code basically consists of two segments
  i) Detects and throws an exception. ii) Catches & handles the exception.
- c++ Exception handling mechanism is basically built upon three keywords, namely, try, throw and catch.

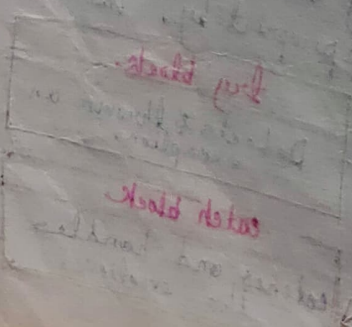
The keyword try is used to preface a block of statements which generate exception. This block of statements is known as try block. When an exception is detected, it is thrown using a throw statement in the try block.

A catch block defined by the keyword catch 'catches' the exception 'thrown' by the throw statement in the try block and handles it appropriately. The relationship is shown below:



Try block throwing an exception:

```
#include <iostream>
using namespace std;
int main ()
{
    int a, b, x;
    cout << "Enter values of a and b\n";
    cin >> a >> b;
    x = a - b
    try
    {
        if ( x != 0)
        {
            cout << "Result (a/x) = " << a/x << endl;
        }
        else          // There is an exception
        {
            throw (x);   // throws int object
        }
    }
    catch (int i)     // Catches the exception
    {
        cout << " Exception caught : DEVIDE BY ZERO\n";
    }
    cout << " End ";
    return 0;
}
```



Call by reference using reference variable

```
#include <iostream.h>
void swap (int &x, int &y)
{
    int z = x;
    x = y;
    y = z;
}
int main()
{
    int a = 45, b = 35
    cout << " Before Swap \n";
    cout << " a = " << a << "b = " << b << " \n";
    Swap (a, b);
    cout << " After swap with pass by reference \n";
    cout << " a = " << a << " b = " << b << "\n";
}
```

```
(int *x, int *y)
{
int z = *x;
x = *y;
y = *z;
}
```

Swap (&a, &b)

after swapping
45 35
later
35 45

Call by Reference using reference variable	Call by reference using pointer variable
i) Here, memory space is ~~create~~ conserved	i) Memory space is not conserve
ii) Notational clearness	iii) Notation clearness is not preserved

"This" pointer holds the base address of the object that has made the recent call to function.

- If data member name and argument name are same, "this" pointer is used.

```
#include <iostream.h>
class Sample
{
  private:
      int i;                    // data member
  public:
      Sample ()
      {
        i=0;
      }
      Sample (int i)
      {
        this->i = i;
      }
};
int main()
{
  Sample A (10)
  return 0;
}
```

data member

argument "i"

Friend Function

```
#include <iostream.h>
using namespace std;
class Sample
{
  private:
        int i;
  public:
        friend void set (Sample);  // friend function is declared not defined within the class
};
void set (Sample A)  // The friend function is defined here the outside of the class
{
  A.i = 10;
}
int main ()
{
   Sample X;
   set (X);   // cannot be called using the object of the class.
   return 0;
}
```

Definition:

A friend function is used to access the private member of a class. The friend function is declared with the keyword **friend**.

Characteristics:

i) Friend function is declared not defined within the class.

ii) Friends " " defined the outside of class.

iii) It can be invoked like a normal function without the help of any object.

iv) It can be declared either the public or private part of a class without effecting its meaning.

v) It has the objects as argument.

Visibility Mode of Access Specifier

The general form of defining a derived class is

~~class base dc~~

```
class derived class name : visibility mode base class name
{
_____ // member of derived class.
};
```

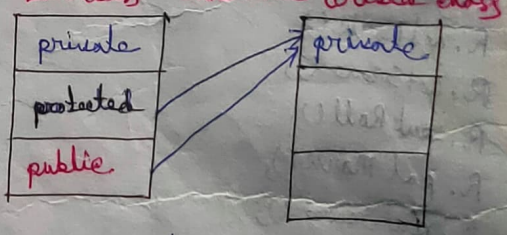
Here the colon(:) indicates that the derived class is derived from the base class name. And the visibility mode is optional, if present, may be either private or protected or public. Remember that the default visibility mode is private. Visibility mode specifies whether the features of base class are privately derived or protectedly derived, or publicly derived

see book 182

~~Public mode~~

Private mode inheritance:

When a base class is privately inherited by a derived class, 'public members' of ~~the base~~ the base class became "private members" of drived class and "protected members" of the base class became as "private members" of derived class. And the "private members" of base class can not be inherited in the derived class.

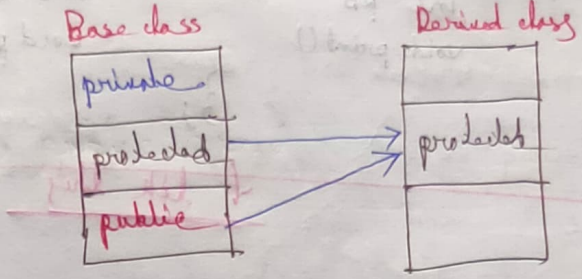


private mode inheritance

(Contd.)

Protected mode inheritance :

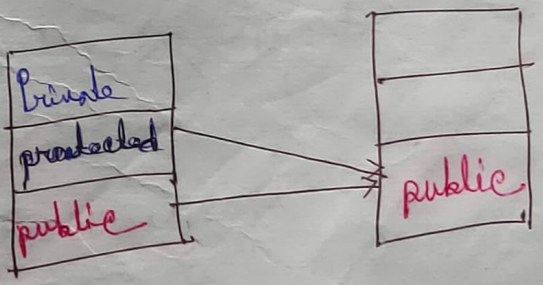
When a base class is protectedly inherited by a derived class, 'public members' of base class become and 'protected members' of base class become as 'protected' of the derived class. And the 'private members' of base class can not be inherited in the derived class.



Protected mode

Public mode inheritance :

When a base class is publicly inherited by a derived class, 'protected member' and 'public member' of base class become as 'private member' of the derived class. And the 'private member' of base class are inheritaned in the derived class.



Public mode.