Department of Computer Science and Engineering

PES UNIVERSITY

Object Oriented Programming with C++

UE22CS221A

UNIT 4

Ms. Kusuma K V

Virtual Functions and Polymorphism, Function Overriding

Basics on upcasting and downcasting.

Upcast: Base class pointer holding derived class object (allowed).

Downcast: Derived class pointer holding base class object (Error. If done forecfully via cast then no compilation error, but risky)

Note: sliced down: Slicing happens when an object of derived type is used to initialize or assign an object of the base type(by valuei.e., not by pointer or reference). The derived portion of the object is "sliced down," leaving only the base portion, which is assigned to the base. Eg: A a=b; (where a is base class object and b is derived class object).

//upCast_StaticPoly.cpp

```
//pgm demonstrates the different ways to access data members and member functions of a class
//1)via object 2)via pointer 3)via reference
//The pgm also demonstrates static polymorphism (none of the member functions are virtual)
#include<iostream>
using namespace std;

class A
{
  int ma1;
  int ma2;
public:
  A(int a1,int a2){ma1=a1;ma2=a2;}
  void display(){cout<<ma1<<""<<ma2<<endl;}
};
class B:public A
{
  int mb1;
public: int mb2;

  B(int a1,int a2,int b1,int b2):A(a1,a2){mb1=b1;mb2=b2;}
  void display(){A::display();cout<<mb1<<""<<mb2<<endl;}
};

int main()
{
  A a(2,3);
  cout<<"Object a"<<endl;
```

```
  a.display();

  B b(4,5,6,7);
  cout<<"Object b"<<endl;
  b.display();

  A* pA=&a;
  B* pB=&b;

  cout<<endl;
  cout<<"Object a via ptr"<<endl;
  pA->display();

  cout<<"Object b via ptr"<<endl;
  pB->display();

  A& rA=a;
  B& rB=b;
  cout<<endl;
  cout<<"Object a via reference"<<endl;
  rA.display();

  cout<<"Object b via reference"<<endl;
  rB.display();


//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class pointer
//        pB=&a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  pA=&b;

  cout<<endl;
  cout<<"Static Polymorphism, even though pA holds derived class object, it calls A::display() ";
  cout<<"because display() is not virtual in base class. ";
  cout<<"Binding happens at compile time, function is called based on pointer type"<<endl;
  pA->display();          //calls A::display(), because static binding happens
```

```
//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class reference
//          rB=a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  A& refA=b;

  cout<<endl;
  cout<<"Static Polymorphism, even though refA holds derived class object, it calls A::display() ";
  cout<<"because display() is not virtual in base class. ";
  cout<<"Binding happens at compile time, function is called based on reference type"<<endl;
  refA.display();          //calls A::display(), because static binding happens

  return 0;
}
/*Output:
Object a
2 3
Object b
4 5
6 7
Object a via ptr
2 3
Object b via ptr
4 5
6 7

Object a via reference
2 3
Object b via reference
4 5
6 7

Static Polymorphism, even though pA holds derived class object, it calls A::display() because display() is not virtual in base class. Binding happens at compile time, function is called based on pointer type
4 5
```

Static Polymorphism, even though refA holds derived class object, it calls A::display() because display() is not virtual in base class. Binding happens at compile time, function is called based on reference type

4 5

*/

Virtual functionis a member function that defines type-specific behavior. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound (Polymorphism).

- A base class specifies that a member function should be dynamically bound by preceding its declaration with the keyword **virtual**.
- Any nonstatic member function, other than a constructor, may be virtual.
- The virtual keyword appears only on the declaration inside the class and may not be used on a function definition that appears outside the class body.
- A function that is declared as virtual in the base class is implicitly virtual in the derived classes as well.

Because the decision as to which version to run depends on the type of the argument, that decision can't be made until run time. Therefore, dynamic binding is sometimes known as run-time binding.

Override: Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.

- A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.
- With one exception, the return type of a virtual in the derived class also must match the return type of the function from the base class. The exception applies to virtuals that return a reference (or pointer) to types that are themselves related by inheritance. That is, if D is derived from B, then a base class virtual can return a B* and the version in the derived can return a D*. However, such return types require that the derived-to-base conversion from D to B is accessible.

Note: In C++, dynamic binding applies only to functions declared as virtual and called through a reference or pointer. Member functions that are not declared as virtual are resolved at compile time, not run time.

Note: A derived-class object contains a subobject corresponding to each of its base classes. Because every derived object contains a base part, we can convert a reference or pointer to a derived-class type to a reference or pointer to an accessible base class.

Note: calls to any function (virtual or not) on an object are also bound at compile time.

//dynamicPoly_Virtual_Override.cpp

//The pgm demonstrates dynamic polymorphism

```
//In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class.
//Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.
#include<iostream>
using namespace std;

class A
{
  int ma1;
  int ma2;
public:

  A(int a1,int a2){ma1=a1;ma2=a2;}
  virtual void display(){cout<<ma1<<""<<ma2<<endl;}                    //virtual function
};

class B:public A
{
  int mb1;
public: int mb2;
  B(int b1,int b2,A a):A(a){mb1=b1;mb2=b2;}
  void display(){A::display();cout<<mb1<<""<<mb2<<endl;}  //A::display() is overridden
};
int main()
{
  A a(2,3);
  cout<<"Object a"<<endl;
  a.display();

  B b(4,5,a);
  cout<<"Object b"<<endl;
  b.display();

  A* pA=&a;
  B* pB=&b;

  cout<<endl;
  cout<<"Object a via ptr"<<endl;
  pA->display();
```

```
  cout<<"Object b via ptr"<<endl;
  pB->display();

  A& rA=a;
  B& rB=b;

  cout<<endl;
  cout<<"Object a via reference"<<endl;
  rA.display();

  cout<<"Object b via reference"<<endl;
  rB.display();

//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class pointer
//        pB=&a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  pA=&b;

  cout<<endl;
  cout<<"Dynamic Polymorphism, pA holds derived class object, it calls B::display() ";
  cout<<"because display() is virtual in base class, binding happens at run time. ";
  cout<<"Function is called depending on the object held by pointer at run time"<<endl;
  pA->display();          //calls B::display(), because dynamic binding happens

//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class reference
//        rB=a;


//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  A& refA=b;

  cout<<endl;
  cout<<"Dynamic Polymorphism, refA holds derived class object, it calls B::display() ";
  cout<<"because display() is virtual in base class, binding happens at run time. ";
  cout<<"Function is called depending on the object held by reference at runtime"<<endl;
  refA.display();         //calls B::display(), because dynamic binding happens
```

```
  return 0;
}

/*
Output:
Object a
2 3
Object b
2 3
4 5

Object a via ptr
2 3
Object b via ptr
2 3
4 5

Object a via reference
2 3
Object b via reference
2 3
4 5

Dynamic Polymorphism, pA holds derived class object, it calls B::display() because display() is virtual in base class, binding happens at run time. Function is called depending on the object held by pointer at run time
2 3
4 5

Dynamic Polymorphism, refA holds derived class object, it calls B::display() because display() is virtual in base class, binding happens at run time. Function is called depending on the object held by reference at runtime
2 3
4 5
*/
```

final and override specifiers (C++ 11 features)

It is legal for a derived class to define a function with the same name as a virtual in its base class but with a different parameter list(function overload). The compiler considers such a

function to be independent from the base-class function. In such cases, the derived version does not override the version in the base class. In practice, such declarations often are a mistake—the class author intended to override a virtual from the base class but made a mistake in specifying the parameter list.

Finding such bugs can be surprisingly hard. Under the new standard we can specify override on a virtual function in a derived class. Doing so makes our intention clear and (more importantly) enlists the compiler in finding such problems for us. The compiler will reject a program if a function marked override does not override an existing virtual function:

```
class B
{
virtual void f1(int) const;
virtual void f2();
void f3();
};
class D1 : public B
{
void f1(int) const override;          // ok: f1 matches f1 in the base
void f2(int) override;          // error: B has no f2(int) function
void f3() override;                   // error: f3 not virtual
void f4() override;                   // error: B doesn't have a function named f4
};
```

In D1, the override specifier on f1 is fine; both the base and derived versions of f1 are const members that take an int and return void. The version of f1 in D1 properly overrides the virtual that it inherits from B.

The declaration of f2 in D1 does not match the declaration of f2 in B—the version defined in B takes no arguments and the one defined in D1 takes an int. Because the declarations don't match, f2 in D1 doesn't override f2 from B; it is a new function that happens to have the same name. Because we said we intended this declaration to be an override and it isn't, the compiler will generate an error.

Because only a virtual function can be overridden, the compiler will also reject f3 in D1. That function is not virtual in B, so there is no function to override.

Similarly f4 is in error because B doesn't even have a function named f4.

We can also designate a function as **final**. Any attempt to override a function that has been defined as final will be flagged as an error:

```
class D2 : public B
{
// inherits f2() and f3() from B and overrides f1(int)
void f1(int) const final;          // subsequent classes can't override f1(int)
```

```
};
class D3 : public D2 {
void f2();                          // ok: overrides f2 inherited from the indirect base, B
void f1(int) const;                 // error: D2 declared f2 as final
};
```

Note: final and override specifiers appear after the parameter list (including any const or reference qualifiers) and after a trailing return.

//multiLvlInh.cpp

```
//Pgm demonstrates multi level inheritance and dynamic polymorphism
//Once a function is defined virtual, it remains virtual down the inheritance hierarchy
#include<iostream>
using namespace std;

class A
{
   int m_a1,m_a2;
public:
A(int a1,int a2)
  {
          m_a1=a1;
          m_a2=a2;
  }
  virtual void display(){cout<<"Base"<<endl;cout<<m_a1<<""<<m_a2<<endl;}
};

class B:public A
{
  int m_b1;

public:
  B(int a1,int a2,int b1):A(a1,a2)
  {
          m_b1=b1;
  }
  void display(){                           //display() is overridden
          A::display();
          cout<<"Derived"<<endl;cout<<m_b1<<endl;
  }
};
```

```
class C:public B
{
  public:C(int a1,int a2,int a3):B(a1,a2,a3){}
  //display is inherited, and not overridden here
};

int main()
{
  A a(2,3);
  B b(4,5,6);
  C c(7,8,9);


  B* pB=&c;
  pB->display();      //Base class pointer holding derived class object c. Calls inherited display() in class C

  cout<<endl;
  A* pA=&b;
  pA->display(); //Base class pointer holding derived class object b. Calls the overridden display() in class B
  return 0;
}
/* Output:
Base
7 8
Derived
9

Base
4 5
Derived
6
*/
```

VTBL and VPTR

To understand how method hiding is avoided, you need to know a bit more about what the virtual keyword actually does. When a class is compiled in C++, a binary object is created that contains all methods for the class. In the non- virtual case, the code to transfer control to theappropriate method is hard-coded directly where the method is called based on the compile-time type.

If the method is declared virtual , the correct implementation is called through the use of a special area of memory called the vtable, for “virtual table.” For each class that has one or more virtual methods there is a vtable, and every object of such a class contains a pointer to said vtable. This vtable contains pointers to the implementations of the virtual methods. In this way, when a method is called on an object, the pointer is followed into the vtable and the appropriate version of the method is executed based on the actual type of the object.

To better understand how vtables make overriding of methods possible, take the following Super and Sub classes as an example.

```
class Super
{
  public:
        virtual void func1() {}
        virtual void func2() {}
        void nonVirtualFunc() {}
};
class Sub : public Super
{
  public:
        virtual void func2() override {}
        void nonVirtualFunc() {}
};
```

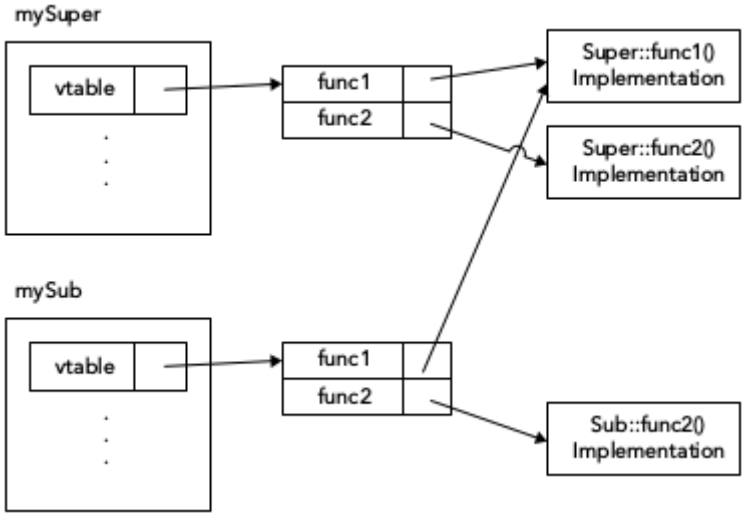
For this example, assume that you have the following two instances:

```
Super mySuper;
Sub mySub;
```

Below figure shows a high-level view of how the vtables for both instances look. The mySuper object contains a pointer (VPTR i.e., Virtual Pointer) to its vtable. This vtable has two entries, one for func1() and one for func2() . Those entries point to the implementations of Super::func1() and Super::func2() .

mySub also contains a pointer (VPTR) to its vtable which also has two entries, one for func1() and one for func2() . The func1() entry of the mySub vtable points to Super::func1() because Sub does not override func1() . On the other hand, the func2() entry of the mySub vtable points to Sub::func2() .

Note that both vtables do not contain any entry for the nonVirtualFunc() method because that method is not virtual.



Whenever a call is dispatched to a virtual function through a reference to an object or a pointer to an object, then the value of the VPTR of the object is obtained. Then the address of the called function from the corresponding VTBL is obtained. Finally, the function is called through the address thus obtained.

Pure Virtual Functions and Abstract Base Class (ABC)

Pure virtual functionis a virtual function declared in the class using **= 0** just before the semicolon. A pure virtual function need not be (but may be) defined. Classes with pure virtuals are **abstract classes**.

The = 0 may appear only on the declaration of a virtual function in the class. It is worth noting that we can provide a definition for a pure virtual. However, the function body must be defined outside the class. That is, we cannot provide a function body inside the class for a function that is = 0.

A class containing (or inheriting without overridding) a pure virtual function is an **abstract base class**. An abstract base class defines an interface for subsequent classes to override. If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well. We cannot create objects of a type that is an abstract class.

Note: A derived class constructor initializes its direct base class only.

//pureVirtual_ABC.cpp

```
//Program demonstrates pure virtual function (Virtual function equated to zero) and
//Abstract Base Class(Class which has a pure virtual function).
//Abstract Base Class is not instantiable.
//Pgm also demonstrates hierarchial inheritance (2 or more derived classes deriving from the same base class)
#include<iostream>
```

```
using namespace std;
class Shape
{
  public:virtual void area()=0;
};
//To show that pure virtual function may have a body. But it has to be defined outside the class.
//void Shape::area(){cout<<"Body"<<endl;}
class Rectangle:public Shape
{
  double ml,mb;
  public: Rectangle(double l=0,double b=0){ml=l;mb=b;}
          void area(){cout<<"Rectangle Area="<<ml*mb<<endl;}

};
class Circle:public Shape
{
  double pi;
  double mr;
  public: Circle(double r=0){mr=r; pi=3.142;}
          void area(){cout<<"Circle Area="<<pi*mr*mr<<endl;}
};
int main()
{
  //Below line is an error. BecauseShape is an Abstract Base Class(ABC) and hence not instantiable.
// Shape s;

  Rectangle r(2,3);
  r.area();

  Circle c(2);
  c.area();

//Calling Base class area by an object of derived class. Uncomment the body of Shape::area to not get linker error
//        r.Shape::area();          //uncomment this line to call base class area i.e., Shape::area

  return 0;
}
```

```
/* Output:
Rectangle Area=6
Circle Area=12.568
*/
```

Virtual Destructors

Non-virtual destructors may result in situations in which memory is not freed by object destruction. If a class is marked as final then destructor may be non- virtual.

For example, if a derived class uses memory that is dynamically allocated in the constructor and deleted in the destructor, it will never be freed if the destructor is never called.

As the following code shows, it is easy to "trick" the compiler into skipping the call to the destructor if it is non- virtual :

```
class Base
{
  public:
        Base() {}
        ~Base() {}
};
class Derived : public Base
{
  public:
        Derived() { mString = new char[30]; }
        ~Derived() { delete [] mString; }
  private:
        char* mString;
};
int main()
{
  Base* ptr = new Derived();          // mString is allocated here.
  delete ptr;          // ~Base is called, but not ~ Derived because the destructor is not virtual!
  return 0;
}
```

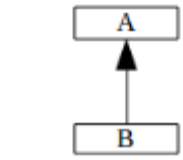
Different types of inheritance:

Single level inheritance (One derived class inherits from one base class)

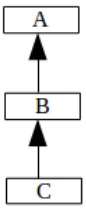
Multi level inheritance (A class inherits from a derived class) Eg: **multiLvlInh.cpp**

Hierarchial inheritance (A single class serves as a base class for more than one derived class) Eg: **pureVirtual_ABC.cpp**

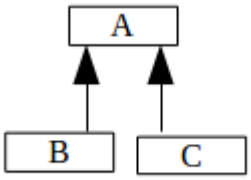
Multiple inheritance (A class derives from more than one base class)



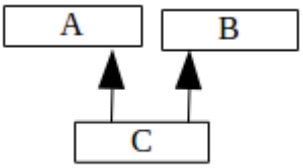
Single Inheritance



Multi level Inheritance



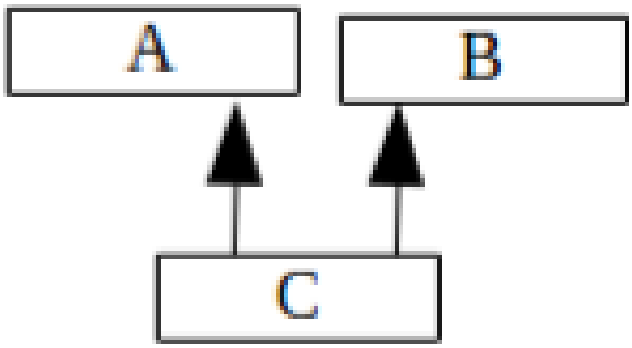
HierarchialInheritance



Multiple Inheritance

Multiple Inheritance, Virtual Base Classes

Multiple Inheritance: A class derives from more than one base class.



Multiple Inheritance

Eg: TA is a Teacher, TA is a Student. Therefore, TA inherits from both Teacher and Student class.

```
class Student{};                          // Base Class = Student
class Teacher{};                          // Base Class = Teacher
class TA: public Student, public Teacher{};     // Derived Class = TA
```

TA inherits properties and operations of both Student as well as Teacher

Note: If all the base classes are to be derived in public then in the derivation list, all base classes should be preceded by the keyword public. Otherwise, the class not preceded with the access specifier will be derived in private as default inheritance for class is private

Eg:
```
class Base1{};
class Base2{};
class Derived: public Base1,Base2{};
```

In the above Eg: Base1 is derived in public whereas Base2 in private

Multiple Inheritance in C++ Semantics

```
class Base1{};
class Base2{};
class Derived: public Base1, public Base2{};
```

- Use keyword public (private, protected) after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- public and private inheritance may be mixed

Derived ISA Base1, Base2

- Data Members
  - Derived class inherits all data members of all Base classes
  - Derived class may add data members of its own
- Member Functions
  - Derived class inherits all member functions of all Base classes
  - Derived class may override a member function of any Base class by redefining it with the same signature
  - Derived class may overload a member function of any Base class by redefining it with the same name; but different signature
- Access Specification
  - Derived class cannot access private members of any Base class
  - Derived class can access protected and public members of any Base class
- Construction-Destruction
  - A constructor of the Derived class must first call constructors of the Base classes to construct the Base class instances of the Derived class – Base class constructors are called in the derivation listing order
  - The destructor of the Derived class must call the destructors of the Base classes to destruct the Base class instances of the Derived class

Multiple Inheritance in C++: Data Members and Object Layout

- Derived ISA Base1, Base2

Data Members

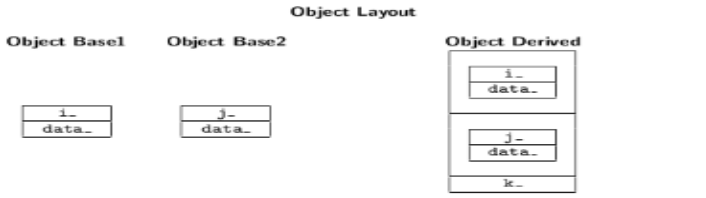
- Derived class inherits all data members of all Base classes
- Derived class may add data members of its own

Object Layout

- Derived class layout contains instances of each Base class
- Further, Derived class layout will have data members of its own

- C++ does not guarantee the relative position of the Base class instances and Derived class members

```
class Base1
{
    protected:
            int i_;
            int data_;
    public: // ...
};

class Base2 {
    protected:
            int j_;
            int data_;
    public: // ...
};

class Derived : public Base1, public Base2
{
            int k_;
    public: // ...
};
```



Multiple Inheritance in C++: Member Functions – Overrides and Overloads

- Derived ISA Base1, Base2
- Member Functions
  - Derived class inherits all member functions of all Base classes
  - Derived class may override a member function of any Base class by redefining it with the same signature
  - Derived class may overload a member function of any Base class by redefining it with the same name; but different signature
- Static Member Functions
  - Derived class does not inherit the static member functions of any Base class
- Friend Functions
  - Derived class does not inherit the friend functions of any Base class

An object of a class defined by multiple inheritance contains not only the data members defined in the derived class, but also data members of all the base classes.

Also, w.r.t such an object, it is possible to call the member functions of not only the derived class, but also the member functions of all the base classes.

//pgm01_multipleInherit.cpp

```
//This program demonstrates that the derived class inherits all the data members and
// all the member functions of all the base classes
#include<iostream>
using namespace std;

class A
{
 public:
        void setX(int x1){x=x1;}
        int getX(){return x;}
 protected:
```

```
        int x;
};

class B
{
 public:
        void setY(int y1){y=y1;}
        int getY(){return y;}
 protected:
        int y;
};
class C: public A, public B
{
 public:
        void setZ(int z1){z=z1;}
        int getZ(){return z;}
 protected:
        int z;
};

int main()
{
        C c1;

        c1.setX(10);
        c1.setY(20);
        c1.setZ(30);

        cout<<"x="<<c1.getX()<<"  y="<<c1.getY()<<"  z="<<c1.getZ()<<endl;
        return 0;
}
/* Output:
x=10  y=20  z=30
*/
```

//pgm02_memberFn_ctor_dtor.cpp

```
//This program demonstrates that derived class inherits all member functions of all base classes
//It may overload or override member fns of Base classes and add its own member fns
//ctor call sequence is in the order of derivation list
```

```
//dtors are called in reverse order of ctor

#include<iostream>
using namespace std;

class Base1
{
 public:
        Base1(int x1){x=x1;cout<<"Base1 ctor"<<endl;}                //ctor of Base1
        void f(){cout<<"I'm f() of Base1"<<endl;}
        ~Base1(){cout<<"Base1 dtor"<<endl;}
 protected:
        int x;
};

class Base2
{
 public:
        Base2(int y1){y=y1;cout<<"Base2 ctor"<<endl;}                //ctor of Base2
        void g(){cout<<"I'm g() of Base2"<<endl;}
        ~Base2(){cout<<"Base2 dtor"<<endl;}
 protected:
        int y;
};

class Derived: public Base1,public Base2
{
 public:
        //ctor of Derived
        Derived(int x1,int y1,int z1):Base1(x1),Base2(y1){z=z1;cout<<"Derived ctor"<<endl;}

        void h(){cout<<"I'm h() of Derived"<<endl;}            //Add own member fn
       void f(int k){cout<<"I'm overloaded Base1::f() in derived"<<endl; }//Overload
Base1::f()
        void g(){cout<<"I'm overridden Base2::g() in derived"<<endl;}//Override Base2::g()
        ~Derived(){cout<<"Derived dtor"<<endl;}

 protected:
        int z;
};
```

```
int main()
{

        Derived d1(2,3,4);      //Base class ctors are excuted in the order
//no matching fn error: coz Base1::f() is overloaded as f(int) in Derived, use :: or call specific fn
//      d1.f();
        d1.g(); //hides Base2::g() by overriding it, to call Base2::g() use ::, as shown after next fn call
        d1.h();          //calls h() of derived
        d1.Base2::g();  //calls Base2::g() using derived class object
        d1.f(2);//calls Derived::f()
        d1.Base1::f();  //calls Base1::f()
        return 0;
}

/*Output:
Base1 ctor
Base2 ctor
Derived ctor
I'm overridden Base2::g() in derived
I'm h() of Derived
I'm g() of Base2
I'm overloaded Base1::f() in derived
I'm f() of Base1
Derived dtor
Base2 dtor
Base1 dtor
*/
```

Ambiguities in Multiple Inheritance

- Identical members in more than one Base class
  - pgm03_mulInherit_ambiguity1.cpp
  - pgm04_mulInherit_ambiguity2_Override.cpp
  - pgm05_mulInherit_ambiguity2_ScopeResoln.cpp
- Diamond Shaped Inheritance
  - pgm06_diamond.cpp
  - pgm07_diamond1.cpp
  - pgm08_diamondInheritance.cpp

- The virtual specifier states a willingness to share a single instance of the named base class within a subsequently derived class. There are no special constraints on a class used as a virtual base class.
- Virtual derivation affects the classes that subsequently derive from a class with a virtual base; it doesn't affect the derived class itself.

//pgm03_mulInherit_ambiguity1.cpp

```
/*This program demonstrates ambiguity resolution when the names of data members in more than one base classes are identical*/
#include<iostream>
using namespace std;

class Base1
{
  public: Base1(int i1,int data1){i=i1;data=data1;}

  protected:
        int i;
     int data;
};

class Base2 {
 public: Base2(int j1,int data1){j=j1;data=data1;}
 protected:
        int j;
     int data;
};

class Derived : public Base1, public Base2
{
 public:
        Derived(int i1,int data1,int j1,int data2,int k1):Base1(i1,data1),Base2(j1,data2)
        {
                k=k1;
        }
//data is there in both Base1 and Base2. Therefore :: is used to resolve ambiguity, if not used gives error
        void display()
        {
```

```
                cout<<"i="<<i<<" Base1data="<<Base1::data<<" j="<<j<<"
        Base2data="<<Base2::data<<" k="<<k<<endl;
        }
 private:
        int k;
};
int main()
{
        Derived d(2,3,4,5,6);
        d.display();
        return 0;
}
/*Output:
i=2 Base1data=3 j=4 Base2data=5 k=6
*/
```

//pgm04_mulInherit_ambiguity2.cpp

```
//This program demonstrates resolving ambiguity when more than one base classes have member functions with the same name
//To resolve ambiguity, either use :: operator while calling fn using derived class object or override the base class member fn
//This pgm uses method overriding to resolve ambiguity
#include<iostream>
using namespace std;

class Base1
{
 public: void show(){cout<<"I'm Base1::show()"<<endl;}
};

class Base2
{
 public: void show(){cout<<"I'm Base2::show()"<<endl;}
};

class Derived:public Base1,public Base2
{
                //show is overridden, comment this line to see the error
 public: void show(){cout<<"I'm overridden show of Derived"<<endl;}
};
```

```
int main()
{
	Derived d;
//if show is not overridden in derived class then the below function call gives an error
	d.show();
	return 0;
}

/*Output:
I'm overridden show of Derived
*/
```

//pgm05_mulInherit_ambiguity2.cpp

```
//This program demonstrates resolving ambiguity when more than one base classes have member functions with the same name
//To resolve ambiguity, either use :: operator while calling fn using derived class object or override the base class member fn
//This pgm uses :: to resolve ambiguity

#include<iostream>
using namespace std;

class Base1
{
 public: void show(){cout<<"I'm Base1::show()"<<endl;}
};

class Base2
{
 public: void show(){cout<<"I'm Base2::show()"<<endl;}
};

class Derived:public Base1,public Base2
{
};

int main()
{
	Derived d;
```

```
//Below function call d.show() is an Error: Ambiguous call to show() using derived class object.
//Compiler is not able to decide whether to call Base1::show() or Base2::show()
//	d.show();

//To resolve the error use :: as shown below
	d.Base1::show();
	d.Base2::show();
	return 0;
}
/*Output:
I'm Base1::show()
I'm Base2::show()
*/
```

//pgm06_diamond.cpp

```
//This program demonstrates that as the inheritance is not made virtual when deriving Teacher or Student, during the construction of TA object, class Person ctor is called twice
//once during Teacher creation and again once during Student creation
//This can be avoided by making the inheritance of Teacher and Student virtual
//look into pgm07_diamond1.cpp
//Note: In the ctor initialization list of derived class, call can be given only to the direct base (parent) class, grandparent class cannot be called if the inheritance is not virtual
//Eg: TA ctor giving call to Person ctor is erroneous as
//Person is not the direct base class of TA
#include<iostream>
using namespace std;
class Person
{
	protected: string mname;
	public:
	Person(string name){cout<<"Person ctor"<<endl;mname=name;}
};
class Student:public Person
{
	protected: string msrn;
	public:
	Student(string name,string srn):Person(name){cout<<"Student ctor"<<endl;msrn=srn;}
};
```

```
class Teacher:public Person
{
        protected: string mid;
                   int msalary;
        public:
        Teacher(string name,string id,int salary):Person(name){cout<<"Teacher
ctor"<<endl;mid=id;msalary=salary;}
};
class TA:public Student,public Teacher
{
        protected: string msubject;
        public:          //You may uncomment Person(name), and see the error
        TA(string name,string srn,string id,int salary,string subject): /* Person(name),*/
Student(name,srn),Teacher(name,id,salary)
        {
                cout<<"TA ctor"<<endl;
                msubject=subject;
        }
        void display()
        {
        //Note:Ambiguity for mname is resolved using Class name and ::
                cout<<Student::mname<<" "<<msrn<<" "<<mid<<" "<<msalary<<"
"<<msubject<<endl;
        }
};
int main()
{
        TA t1("abc","01PES","01PES",10000,"C++");
        t1.display();
        return 0;
}
/*Output:
Person ctor
Student ctor
Person ctor
Teacher ctor
TA ctor
abc 01PES 01PES 10000 C++
*/
```

```
//pgm07_diamond1.cpp
//This program demonstrates virtual inheritance, which makes class Person a virtual base class in the inheritance hierarchy
//During TA object creation, note that neither Teacher ctor nor Student ctor gives call to Person ctor instead TA class itself give call to Person ctor (Virtual Base Class)

//Note the order of ctors execution (in general):
//1)Virtual Base Class ctors in the order of inheritance
//2)Non Virtual Base Class ctors in the order of inheritance
//3)Member object's ctors in the order of declaration
//4)Derived class ctor
#include<iostream>
using namespace std;

class Person
{
        protected: string mname;
        public:
        Person(string name){cout<<"Person ctor"<<endl;mname=name;}
};
class Student:virtual public Person
{
        protected: string msrn;
        public:
        Student(string name,string srn):Person(name){cout<<"Student
ctor"<<endl;msrn=srn;}
};
class Teacher:virtual public Person
{
        protected: string mid;
                                int msalary;
        public:
        Teacher(string name,string id,int salary):Person(name){cout<<"Teacher
ctor"<<endl;mid=id;msalary=salary;}
};
class TA:public Student,public Teacher
{
        protected: string msubject;
        public:
```

```
        TA(string name,string srn,string id,int salary,string
subject):Person(name),Student(name,srn),Teacher(name,id,salary)
        {
                cout<<"TA ctor"<<endl;
                msubject=subject;
        }
        void display()
        {
                cout<<mname<<" "<<msrn<<" "<<mid<<" "<<msalary<<" "<<msubject<<endl;
        }
};
int main()
{
        TA t1("abc","01PES","01PES",10000,"C++");
        t1.display();
        return 0;
}
/*
Output:
Person ctor
Student ctor
Teacher ctor
TA ctor
abc 01PES 01PES 10000 C++
*/
```

//pgm08_diamondInheritance.cpp

```
#include<iostream>
using namespace std;

class A
{
 protected: int a;
 public:
        A(int a1){a=a1;cout<<"ctor A"<<endl;} A(){cout<<"default ctor A"<<endl;}
        ~A(){cout<<"dtor A"<<endl;}
};
```

```
class B:virtual public A
{
  protected: int b;
  public:
        B(int a1,int b1):A(a1){b=b1;cout<<"ctor B"<<endl;}
        ~B(){cout<<"dtor B"<<endl;}
};
class C:virtual public A
{
  protected: int c;
  public:
        C(int a1,int c1):A(a1){c=c1;cout<<"ctor C"<<endl;}
        ~C(){cout<<"dtor C"<<endl;}
};
class D:public B,public C
{
  protected: int d;
  public:
        D(int a1,int b1,int c1,int d1):A(a1),B(a1,b1),C(a1,c1){d=d1;cout<<"ctor D"<<endl;}
        ~D(){cout<<"dtor D"<<endl;}
        void show(){cout<<"a="<<B::a<<" b="<<b<<" c="<<c<<" d="<<d<<endl;}
};
int main()
{
        D d(1,2,3,4);
        d.show();
        return 0;
}
/*Output:
ctor A
ctor B
ctor C
ctor D
a=1 b=2 c=3 d=4
dtor D
dtor C
dtor B
dtor A
*/
```

Type Casting, Run Time Type Identification (RTTI)

A cast operator takes an expression of source type (implicit from the expression) and convert it to an expression of target type (explicit in the operator) following the semantics of the operator.

operator<type>(value whose type is to be converted)

C++ Style cast operators

a) const_cast
b) dynamic_cast
c) static_cast
d) reinterpret_cast

const_cast operator is used to take away the constness of a variable.

//pgm09_constCast1.cpp

```
#include <iostream>
using namespace std;

void print(char * str) { cout << str<<endl; }

int main()
{
        const char *c = "sample text";
       // print(c);  // error: ’void print(char *)’: cannot convert argument 1 from ’const char *’ to ’char *’

        print(const_cast<char *>(c));    //Removes the constness associated with 'c' and passes to print()
        return 0;
}
/*Output:
sample text
*/
```

//pgm10_constCast2.cpp

```
#include <iostream>
using namespace std;

class A
{
        int i;
public:
```

```
        A(int i1) : i(i1) {}
        int get() const { return i; }
        void set(int i1) { i = i1; }
};

int main()
{
        const A a(1);
        cout<<a.get()<<endl;
        //a.set(5);   // error: 'void A::set(int)': cannot convert 'this' pointer from 'const A' to
'A &'
        const_cast<A&>(a).set(5);
        //const_cast<A>(a).set(5);      // error: 'const_cast': cannot convert from 'const A' to
'A' i.e., cannot strip the constness of the object, so as given in the above line create a non
const reference to the object and pass it to the function
        cout<<a.get()<<endl;

        return 0;
}
/* Output:
1
5
*/
```

static_cast can be used to perform explicit conversions that are supported directly by the language.

For example, if you write an arithmetic expression in which you need to convert an int to a double in order to avoid integer division, use a static_cast.

In this example, it's enough to only static_cast i, because that makes one of the two operands a double, making sure C++ performs floating point division.

```
int i = 3;
int j = 4;
double result = static_cast<double>(i) / j;
```

//pgm11_staticCast.cpp

```
#include <iostream>
using namespace std;
// Built-in Types
int main() {
        int i = 2;
```

```
        double d = 3.7;
//      double *pd = &d;

        i=d;                            // implicit --
        i = static_cast<int>(d);// static_cast -- okay
        i=(int)d;                       // C-style -- okay
        cout<<i<<endl;

        d = i;                          // implicit -- okay
        d = static_cast<double>(i);     // static_cast -- okay
        d = (double)i;                  // C-style -- okay
        cout<<d<<endl;
        return 0;
}
/*Output:
3
3
*/
```

//pgm12_staticCast2.cpp

```
#include <iostream>
using namespace std;
// Class Hierarchy
class A { };
class B: public A { };
int main()
{
        A a;
        B b;

        // UPCAST
        A *p = &b;                      // implicit -- okay
        p = static_cast<A*>(&b);        // static_cast -- okay
        p = (A*)&b;                     // C-style -- okay

        // DOWNCAST
//      B* q = &a;                      // implicit -- error
        B* q = static_cast<B*>(&a);   // static_cast -- okay: RISKY: Should use dynamic_cast
        q = (B*)&a;                     // C-style -- okay
        return 0;
}
```

Another use for the static_cast is to perform downcasts in an inheritance hierarchy. For example:

//pgm13_staticCast3.cpp

```
#include<iostream>
using namespace std;
class Base
{
  protected:
        int mb=10;
  public:
        Base() {}
        virtual ~Base() {}
        void display()
        {
              cout<<"Base::display(): "<<mb<<" ";
        }
};
class Derived : public Base
{
  protected:
        int md=20;
  public:
        Derived() {}
        virtual ~Derived() {}
        void display()
        {
              Base::display();
              cout<<"Derived::display(): "<<md<<endl;
        }
};
int main()
{
  Base* pb=new Base();
  Derived* pd = new Derived();
  cout<<"Display before typecast"<<endl;
  pb->display();
  cout<<endl;
  pd->display();

  delete pb;
```

```
  delete pd;

  Base b; Derived d;
  pb = &d;                          // Don't need a cast to go up the inheritance hierarchy
  pd = static_cast<Derived*>(&b);   // Need a cast to go down the hierarchy, RISKY

  cout<<endl<<"Base pointer pointing to derived object"<<endl;
  pb->display();
  cout<<endl;
  cout<<"Derived pointer pointing to base object"<<endl;
  cout<<"Junk or Program may crash, so better to use dynamic_cast(uses RTTI) and check return value before operation"<<endl;
  pd->display();

  cout<<endl<<"Using reference for static_cast"<<endl;
  Base base;      Derived derived;
  Base& br = base;
  Derived& dr = static_cast<Derived&>(br);
  br.display();
  cout<<endl;
  cout<<"Junk or Program may crash, so better to use dynamic_cast(uses RTTI) and use exception Handling"<<endl;
  dr.display();
  return 0;
  }
```

These casts work with both pointers and references. They do not work with objects themselves. Note that these casts with static_cast do not perform run-time type checking. They allow you to convert any Base pointer to a Derived pointer or Base reference to a Derived reference, even if the Base really isn't a Derived at run time.

For example, the following code will compile and execute, but using the pointer d can result in potentially catastrophic failure, including memory overwrites outside the bounds of the object.

```
Base* b = new Base();
Derived* d = static_cast<Derived*>(b);
```

To perform the cast safely with run-time type checking, use the dynamic_cast.

dynamic_cast and RTTI(Run Time Type Information): The dynamic_cast provides a run-time check on casts within an inheritance hierarchy.

- You can use it to cast pointers or references.

- dynamic_cast checks the run-time type information of the underlying object at run time. If the cast doesn't make sense, dynamic_cast returns a null pointer (for the pointer version) or throws an std::bad_cast exception (for the reference version).
- Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.
- Note that the run-time type information is stored in the vtable of the object. Therefore, in order to use dynamic_cast , your classes must have at least one virtual method. If your classes don't have a vtable, trying to use dynamic_cast will result in a compiler error. (source type is not polymorphic). However, if you are upcasting on non-polymorphic hierarchy, compiler doesn't show error.

//pgm14_dynamicCast.cpp

```
#include<iostream>
using namespace std;
class Base
{
        protected:
                int mb=10;
        public:
                Base() {}
                virtual ~Base() {}
                void display()
                {
                        cout<<"Base::display(): "<<mb<<" ";
                }
};
class Derived : public Base
{
        protected:
                int md=20;
        public:
                Derived() {}
                virtual ~Derived() {}
                void display()
                {
                        Base::display();
                        cout<<"Derived::display(): "<<md<<endl;
                }
};
int main()
{
```

```
        Base* pb=new Base();
        Derived* pd = new Derived();
        cout<<"Display before typecast"<<endl;
        pb->display();
        cout<<endl;
        pd->display();

        delete pb;
        delete pd;

        Base b; Derived d;
        pb = dynamic_cast<Base*>(&d);       //However, don't need a cast to go up the inheritance hierarchy
        pd = dynamic_cast<Derived*>(&b); // Need a cast to go down the hierarchy, RISKY
        cout<<endl<<"Base pointer pointing to derived object"<<endl;
        if(pb)
        {
                pb->display();
                cout<<endl;
        }
        else
                cout<<"dynamic_cast returned NULL"<<endl;

        cout<<endl<<"Derived pointer pointing to base object"<<endl;
        if(pd)
        {
                pd->display();
                cout<<endl;
        }
        else
                cout<<"dynamic_cast returned NULL"<<endl;

        cout<<endl<<"Using reference for dynamic_cast"<<endl;
        Base base;
        Derived derived;
        try
        {
                cout<<"Base reference referring to derived object"<<endl;
                Base& br = dynamic_cast<Base&>(derived);
                br.display();
```

```
            cout<<endl;
        }catch(const bad_cast&)
        {
            cout<<"Bad cast!!"<<endl;
        }

        try{
            cout<<"Derived reference referring to base object"<<endl;
            Derived& dr = dynamic_cast<Derived&>(base);
            dr.display();
        }catch(const bad_cast&)
        {
            cout<<"Bad cast!!"<<endl;
        }
        return 0;
}
/* Output:
Display before typecast
Base::display(): 10
Base::display(): 10 Derived::display(): 20

Base pointer pointing to derived object
Base::display(): 10

Derived pointer pointing to base object
dynamic_cast returned NULL

Using reference for dynamic_cast
Base reference referring to derived object
Base::display(): 10
Derived reference referring to base object
Bad cast!!
*/
```

Note that you can perform the same casts down the inheritance hierarchy with a static_cast or reinterpret_cast . The difference with dynamic_cast is that it performs run-time (dynamic) type checking, while static_cast and reinterpret_cast will perform the casting even if they are erroneous.

reinterpret_cast : reinterpret cast converts any pointer type to any other pointer type, even of unrelated classes.

In theory, you could also use reinterpret_cast to cast pointers to ints and ints to pointers, but this is considered erroneous programming, because on many platforms (especially 64-bit platforms) pointers and int s are of different sizes. For example, on a 64-bit platform, pointers are 64 bit, but integers could be 32 bit. Casting a 64-bit pointer to a 32-bit integer will result in losing 32 critical bits!

Summary of Casts

The following table summarizes the casts you should use for different situations.

SITUATION	CAST
Remove `const`-ness	`const_cast`
Explicit cast supported by language (e.g., `int` to `double`, `int` to `bool`)	`static_cast`
Explicit cast supported by user-defined constructors or conversions	`static_cast`
Object of one class to object of another (unrelated) class	Can't be done
Pointer-to-object of one class to pointer-to-object of another class in the same inheritance hierarchy	`dynamic_cast` recommended, or `static_cast`
Reference-to-object of one class to reference-to-object of another class in the same inheritance hierarchy	`dynamic_cast` recommended, or `static_cast`
Pointer-to-type to unrelated pointer-to-type	`reinterpret_cast`
Reference-to-type to unrelated reference-to-type	`reinterpret_cast`
Pointer-to-function to pointer-to-function	`reinterpret_cast`

Composition:

a) Combining existing objects to create another, more complex object is called composition.

b) When you compose a new object, you create complex behaviour by delegating tasks to the internal objects.

c) Composition is different from inheritance, which defines a new object by changing or refining the behaviour of an old object.

d) Inheritance is suitable only when classes are in a relationship in which subclass is a (kind of) superclass. For example: A Car **is a** Vehicle so the class Car has all the features of class Vehicle in addition to the features of its own class. However, we cannot always have. is a relationship between objects of different classes. For example: A car is not a kind of engine. But a car **has a**n engine (So use composition).

```
//pgm15_composition.cpp
#include<iostream>
using namespace std;
class Date
{
        int day;
        int month;
        int year;
public:
        Date(int dd, int mm ,int yy)
        {
 //       cout<<"Constructor of Date Class";
         day=dd;
         month=mm;
         year=yy;
        }
//Default Constructor of Date Class
        Date(){}
        void display(){cout<<day<<"-"<<month<<"-"<<year<<endl;}
};

class Employee {
        int id;
        string name;
        Date hireDate;          //object of Date class
public:
        Employee(int num, string n, Date hire)
        {
//              cout<<"Constructor of Employee Class is Called"<<endl;
                id = num;
                name = n;
                hireDate = hire;
        }
        void display()
        {
           cout<<"id=" <<id<<" Name="<<name<<" Hiredate=";
                hireDate.display();
        }
};
```

```
int main()
{
        Date d(12, 11, 2020);
        Employee emp(1, "abc", d);
        emp.display();

        return 0;
}
/*Output:
id=1 Name=abc Hiredate=12-11-2020
*/
```

Class templates

Class templates define a class where the types of some of the variables, return types of methods, and/or parameters to the methods are specified as **parameters.**

Class templates are useful primarily for containers, or data structures, that store objects. Classes like list, queue etc.

The data members and the methods are almost the same for list of numbers, list of objects. Yet, we need to define different classes

```
template <class T>
class classname
{

};
```

Here T will be specified when the object of the class is created.

More than one generic data type can also be defined using a comma separated list of parameters.

```
class Stack {
    char data_[100];            // Has type
    int top_;
public:
    Stack() :top_(-1) {}
    ~Stack() {}

    void push(const char& item) // Has type
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const char& top() const     // Has type
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of char

```
class Stack {
    int data_[100];             // Has type
    int top_;
public:
    Stack() :top_(-1) {}
    ~Stack() {}

    void push(const int& item)  // Has type
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const int& top() const      // Has type
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

- Stack of int
- Can we combine these Stack codes using a type variable T?

```
//pgm16_templateStack.cpp
#include<iostream>
using namespace std;

template<class  T>
class Stack
{
        T data[100];
        int top;
 public:
        Stack():top(-1){}
        ~Stack(){}

        void push(const  T& item){data[++top]=item;}
        void pop(){--top;}
        const  T& ptop() const {return data[top];}
        bool empty() const {return top== -1;}
};

int main()
{
        Stack<char> s1;
        s1.push('a');    cout<<s1.ptop()<<endl;
        s1.push('b');    cout<<s1.ptop()<<endl;
        s1.pop();        cout<<s1.ptop()<<endl;

        Stack<int> s2;
        s2.push(20);     cout<<s2.ptop()<<endl;
        s2.push(10);     cout<<s2.ptop()<<endl;
        s2.pop();        cout<<s2.ptop()<<endl;
        return 0;
}
/*Output:
a
b
a
20
10
20
*/
```

```
//pgm17_classTemplate.cpp
#include<iostream>
using namespace std;

template<class T1,class T2>
class Sample
{
        T1 a;
        T2 b;

  public:
        void setData()
        {
                cin>>a>>b;
        }

        void display()
        {
                cout<<"a="<<a<<" b="<<b<<endl;
        }
};

int main()
{
        Sample <int,int> s1;
        Sample <int,char> s2;
        Sample <int,float> s3;

        cout<<"Enter 2 integers"<<endl;
        s1.setData();

        cout<<"Enter an integer and character"<<endl;
        s2.setData();

        cout<<"Enter an integer and float"<<endl;
        s3.setData();

        cout<<"s1 s2 s3:"<<endl;
        s1.display();
        s2.display();
```

```
        s3.display();

        return 0;
}

/*Output:
Enter 2 integers
10 20
Enter an integer and character
30 a
Enter an integer and float
40 45.0
s1 s2 s3:
a=10 b=20
a=30 b=a
a=40 b=45
*/
```

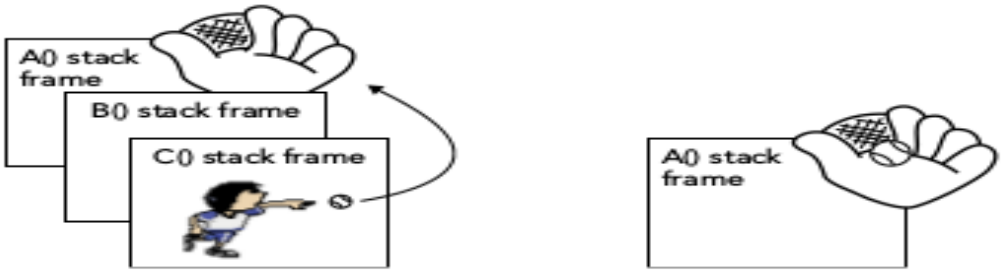
ERRORS AND EXCEPTIONS

- No program exists in isolation; they all depend on external facilities such as interfaces with the operating system, networks and file systems, external code such as third-party libraries, and user input.
- Each of these areas can introduce situations which require responding to problems which may be encountered. These potential problems can be referred to with the general term exceptional situations. Even perfectly written programs encounter errors and exceptional situations.
- Thus, anyone who writes a computer program must include error-handling capabilities. Some languages, such as C, do not include many specific language facilities for error handling.
- Programmers using these languages generally rely on return values from functions and other ad hoc approaches. Other languages, such as Java, enforce the use of a language feature called exceptions as an error-handling mechanism.
- C++ lies between these extremes. It provides language support for exceptions, but does not require their use. However, you can't ignore exceptions entirely in C++ because a few basic facilities, such as memory allocation routines, use them.

What Are Exceptions, Anyway?

- Exceptions are a mechanism for a piece of code to notify another piece of code of an "exceptional" situation or error condition without progressing through the normal code paths.
- The code that encounters the error throws the exception, and the code that handles the exception catches it. Exceptions do not follow the fundamental rule of step-by-step execution to which you are accustomed.

- When a piece of code throws an exception, the program control immediately stops executing code step by step and transitions to the exception handler, which could be anywhere from the next line in the same function to several function calls up the stack.
- If you like sports analogies, you can think of the code that throws an exception as an outfielder throwing a baseball back to the infield, where the nearest infielder (closest exception handler) catches it.
- Below figure(on left) shows a hypothetical stack of three function calls. Function A() has the exception handler. It calls function B() , which calls function C() , which throws the exception. Figure (on right) shows the handler catching the exception. The stack frames for C() and B() have been removed, leaving only A().



Why Exceptions in C++ Are a Good Thing

- As mentioned earlier, run-time errors in programs are inevitable. Despite that fact, error handling in most C and C++ programs is messy and ad hoc. The de facto C error-handling standard, which was carried over into many C++ programs, uses **integer function return codes** and the **errno macro** to signify errors.
- Each thread has its own errno value. errno acts as a thread-local integer variable that functions can use to communicate errors back to calling functions.
- Unfortunately, the integer return codes and errno are used inconsistently.
  - Some functions might choose to return 0 for success and –1 for an error. If they return –1, they also set errno to an error code.
  - Other functions return 0 for success and nonzero for an error, with the actual return value specifying the error code. These functions do not use errno .
  - Still others return 0 for failure instead of for success, presumably because 0 always evaluates to false in C and C++.
- These inconsistencies can cause problems because programmers encountering a new function often assume that its return codes are the same as other similar functions. That is not always true.
- On Solaris 9, there are two different libraries of synchronization objects: the POSIX version and the Solaris version.
- The function to initialize a semaphore in the POSIX version is called sem_init() , and the function to initialize a semaphore in the Solaris version is called sema_init() .
- As if that weren't confusing enough, the two functions handle error codes differently!
  - sem_init() returns –1 and sets errno on error

- sema_init() returns the error code directly as a positive integer, and does not set errno .

Another problem is that the return type of functions in C++ can only be of one type, so if you need to return both an error and a value, you must find an alternative mechanism.

One solution is to return a std::pair or std::tuple , an object that you can use to store two or more types (available in STL).

Another choice is to define your own struct or class that contains several values, and return an instance of that struct or class from your function.

Yet another option is to return the value or error through a reference parameter or to make the error code one possible value of the return type, such as a nullptr pointer.

In all these solutions, the **caller is responsible** to explicitly check for any errors returned from the function and if it doesn't handle the error itself, it should propagate the error to its caller. Unfortunately, this will often result in the loss of critical details about the error.

C programmers may be familiar with a mechanism known as setjmp()/longjmp() . This mechanism cannot be used correctly in C++, because it bypasses scoped destructors on the stack. You should avoid it at all cost, even in C programs

- Exceptions provide an easier, more consistent, and safer mechanism for error handling. There are several specific advantages of exceptions over the ad hoc approaches in C & C++.
- When return codes are used as an error reporting mechanism, you might forget to check the return code and properly handle it either locally or by propagating it upwards. Exceptions cannot be forgotten or ignored: If your program fails to catch an exception, it will terminate.
- When integer return codes are used, they generally do not contain sufficient information. You can use exceptions to pass as much information as you want from the code that finds the error to the code that handles it.
- Exception handling can skip levels of the call stack. That is, a function can handle an error that occurred several function calls down the stack, without error-handling code in the intermediate functions. Return codes require each level of the call stack to clean up explicitly after the previous level.

Exception handling is not enforced in C++.

- For example, in Java a function that does not specify a list of possible exceptions that it can throw is not allowed to throw any exceptions.
- In C++, a function that does not specify a list of exceptions can throw any exception it wants!

Throwing and Catching Exceptions

Using exceptions consists of providing two parts in your program: a try/catch construct, to handle an exception, and a throw statement, that throws an exception. Both must be present in some form to make exceptions work. However, in many cases, the throw happens deep inside some library and the programmer never sees it, but still has to react to it using a try/catch construct.

The try/catch construct looks as follows:

```
try {
// ... code which may result in an exception being thrown
} catch (exception-type1 exception-name) {
// ... code which responds to the exception of type 1
} catch (exception-type2 exception-name) {
// ... code which responds to the exception of type 2
}
// ... remaining code
```

- The code which may result in an exception being thrown might contain a throw directly, or might be calling a function which either directly throws an exception or calls, by some unknown number of layers of calls, a function which throws an exception.
- If no exception is thrown, the code in the catch blocks is not executed, and the "remaining code" which follows will follow the last statement executed in the try block.
- If an exception is thrown, any code following the throw or following the call which resulted in the throw is not executed, but control immediately goes to the **right catch block** depending on the type of the exception that is thrown.
- If the catch block does not do a control transfer, for example
  - by returning a value
  - throwing a new exception or rethrowing the exception,

  then the "remaining code" is executed after the last statement of that catch block.
- The simplest example to demonstrate exception handling is avoiding divide-by-zero.

//pgm18_safeDivide.cpp

```
#include<iostream>
using namespace std;
int safeDivide(int num,int den)
{
        if(den==0)
                throw "div by 0";
        return num/den;
}
int main()
{
```

```
        try {
        cout<<safeDivide(4,2)<<endl;
        cout<<safeDivide(4,0)<<endl;
        cout<<"I'm nvr executed"<<endl;
        }catch(const char *c)
        {
                cout<<"Exception raised: "<<c<<endl;
        }
        return 0;
}
/*Output:
2
Exception raised: div by 0
*/
```

- pgm19_tryCatchThrow.cpp
- pgm20_excepLevel.cpp
- pgm21_rethrow.cpp
- pgm22_multiCatch.cpp
- pgm23_matchAny.cpp
- pgm24_ctorDtorException.cpp
- pgm25_exception_inheritance.cpp
- pgm26_exceptionObj.cpp
- Note: Stack unwinding means that the destructors for all locally-scoped names are called and all code remaining in each function past the current point of execution is skipped.
- However, in stack unwinding, pointer variables are not freed.

//pgm19_tryCatchThrow.cpp

```
#include<iostream>
using namespace std;

int main()
{
//if stmts throwing exceptions are not within try catch block, and exception occurs then it leads to abnormal program termination, uncomment the below throw stmt and check
//        throw 5;

        try{
                throw 5;
                cout<<"i'm never executed here :-("<<endl;
```

```
        }catch(int a)
        {
                cout<<"exception occured, exception no. is "<<a<<endl;
        }
        return 0;
}
/* Output:
exception occured, exception no.is 5
*/
```

//pgm20_excepLevel.cpp

```
#include<iostream>
using namespace std;
void g()
{
        throw "g threw me";
}
void f()
{
        g();
}

int main()
{
        try
        {
                f();
        }catch(char const* c)
        {
                cout<<"caught in main, f didn't handle me: "<<c<<endl;
        }

        return 0;
}
/*Output:
caught in main, f didn't handle me: g threw me
*/
```

//pgm21_rethrow.cpp

```
#include<iostream>
using namespace std;

void g()
{
	throw "g threw me";
}
void f()
{
	try{
		g();
	}catch(const char* c)
	{
		cout<<"caught in f:"<<c<<endl;
		throw(c);	//or just throw, without any arguments causes rethrow
	}

}

int main()
{
	try
	{
		f();
	}catch(const char* c)
	{
		cout<<"caught in main, rethrown from f:"<<c<<endl;
	}
	return 0;
}

/*Output:
caught in f:g threw me
caught in main, rethrown from f:g threw me
*/
```

//pgm22_multiCatch.cpp

```
#include<iostream>
using namespace std;

int main()
{
		try
		{
			throw 5.5;
//Demonstrate replacing 5.5 by 'a', 5, 5.5f, "ab"
		}catch(int a)
		{
			cout<<"I catch int excep: value="<<a<<endl;
		}catch(double a)
		{
			cout<<"I catch double excep: value="<<a<<endl;
		}catch(char a)
		{
			cout<<"I catch char excep: value="<<a<<endl;
		}catch(double a)
		{
			cout<<"I catch double excep: value="<<a<<endl;
		}catch(...)
		{
			cout<<"I catch any"<<endl;
		}
		cout<<"Bye"<<endl;
		return 0;
}
/* Output:
I catch double excep: value=5.5
Bye
*/
```

//pgm23_matchAny.cpp

```
#include<iostream>
using namespace std;
int main()
{
	try{
```

```
                throw 'a';
                cout<<"i'm never executed here :-("<<endl;
        }
        catch(double a)
        {
                cout<<"exception occured, exception is double "<<a<<endl;
        }catch(...)
        {
                cout<<"I match any exception"<<endl;
        }
//Demonstrate by replacing 'a' by 5, 5.5f, "ab", 5.5
        return 0;
}
/*Output:
I match any exception
*/
```

//pgm24_ctorDtorException.cpp

```
//When exception is thrown, the destructors for all locally-scoped names are called and
//all code remaining in each function past the current point of execution is skipped before control goes to catch

#include<iostream>
using namespace std;

class Demo
{
        int a;
        public:
        Demo(int a1)
        {

                a=a1;
        }

        ~Demo(){cout<<"Dtor "<<a<<endl;}
};
```

```
void f3()
{
	Demo a(30);
	throw "f3 threw me";
}

void f2()
{
	Demo a(20);
	f3();
}
void f1()
{
	Demo a(10);
	f2();
}

int main()
{
	try
	{
		Demo a(5);
		f1();
	}catch(const char* str)
	{
		cout<<"Caught in main: "<<str<<endl;
	}
	return 0;
}

/*Output:
Dtor 30
Dtor 20
Dtor 10
Dtor 5
Caught in main: f3 threw me
*/
```

//pgm25_exception_inheritance.cpp

/*When more than one catch clause is used, the catch clauses are matched in syntactic order as they appear in your code; the first one that matches, wins.

If one catch is more inclusive than a later one, it will match first, and the more restrictive one, which comes later, will not be executed at all.

Therefore,catch clauses must be placed from most restrictive to least restrictive order.

In this example, as the base class object may be assigned with derived class object (but slicing happens), the derived class exception is never called */

```
#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
//Uncomment and see the output
/*      Derived d;
        try {
                throw d;
        }
        catch(Base b) {
                cout<<"Caught Base Exception"<<endl;
        }
        catch(Derived d) {  //This catch block is NEVER executed
     cout<<"Caught Derived Exception"<<endl;
        }

 */
//correct way would be as follows
        Base b;
        try {
                throw b;
        }
        catch(Derived d) {
                cout<<"Caught Derived Exception"<<endl;
        }
        catch(Base b) {
```

```
                cout<<"Caught Base Exception"<<endl;
        }
  return 0;
}
/* Output:
Caught Base Exception
*/
```

//pgm26_exceptionObj.cpp

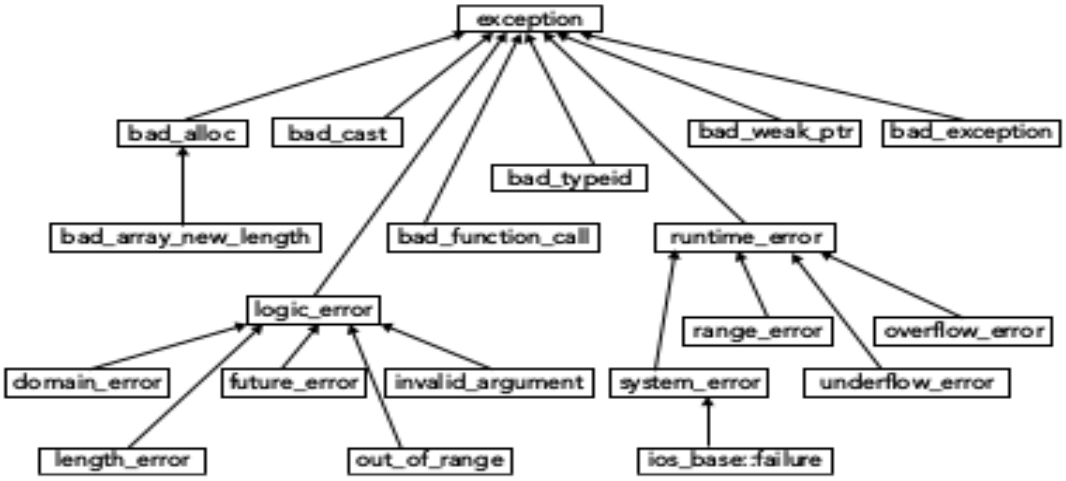
```
#include<iostream>
using namespace std;

int SafeDivide(int num, int den)
{
        if (den == 0)
                throw invalid_argument("Divide by zero");
        return num / den;
}

int main()
{
//Operation which would result in exception, if used outside try catch block then
//if an exception is raised, it will lead to program termination
//      cout<<(10/0)<<endl;
//      cout<<SafeDivide(10,0)<<endl;
        try {
                cout << SafeDivide(5, 2) << endl;
                cout << SafeDivide(10, 0) << endl;
                //below line will never be called, because the previous line throws an
exception, so control goes to catch block
                cout << SafeDivide(3, 3) << endl;
        } catch (const invalid_argument& e) {
                cout << "Caught exception: " << e.what() << endl;
        }

        return 0;
}
/*Output:
```

```
2
Caught exception: Divide by zero
*/
```

The above example throws an exception of type std::invalid_argument.

- throw is a keyword in C++, and is the only way to throw an exception.
- The invalid_argument() part of the throw line means that you are constructing a new object of type invalid_argument to throw. It is one of the standard exceptions provided by the C++ Standard Library. All Standard Library exceptions form a hierarchy. Each class in the hierarchy supports a what() method that returns a const char* string describing the exception. This is the string you provide in the constructor of the exception.

The Standard Exception Hierarchy



- All of the exceptions thrown by the C++ Standard Library are objects of classes in this hierarchy.
- Each class in the hierarchy supports a what() method that returns a const char* string describing the exception. You can use this string in an error message.
- When more than one catch clause is used, the catch clauses are matched in syntactic order as they appear in your code; the first one that matches, wins. If one catch is more inclusive than a later one, it will match first, and the more restrictive one, which comes later, will not be executed at all. Therefore, you should place your catch clauses from most restrictive to least restrictive order.

References:

- "C++ Primer", Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition
- https://nptel.ac.in/courses/106/105/106105151/