1. VIRTUAL FUNCTIONS AND POLYMORPHISM

What is Polymorphism?

Polymorphism means having multiple forms of one thing. Here, one form represents original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes. In inheritance, polymorphism is done by method overriding, when both super and sub class have member function with same declaration but different definition.

Polymorphism can be classified into two types,

1. Compile Time Polymorphism
2. Run Time Polymorphism

Compile time polymorphism can be achieved in C++ by overloading mechanisms. Whereas, run time polymorphism in C++ is achieved using **virtual functions**. Virtual functions are explored more below.

1.1 VIRTUAL FUNCTIONS

When member functions in both base class and the derived class have the same name, the function in the base class is declared as virtual using the keyword **virtual** preceding its normal declaration.

In other words, a virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

When accessed "normally", virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer.

A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon the type of object pointed to by the pointer. And this determination is made at run time. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

In other words, when a function is declared as virtual, the compiler invokes the function at run time depending on the type of object pointed by the base pointer. Now it does not depend on the type of pointer used.

Base class pointer can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked. But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.

Important Tips:

A base pointer can be made to point to any number of derived objects, it cannot access the members defined by a derived class. It can access only the members which are common to the base class. If a same function is present in both base and derived class, always base version of the function is called when we access the function using base pointer (no matters whether it points to base class or derived class). Derived version of the function can be called by making the function (having same name) as virtual. This is also called function overriding because function in the base class is overridden by the function in the derived class.

Remember:

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

To begin, here is a simple example:

Example 1:

```
#include <iostream>
using namespace std;
class base
{
public:
virtual void show()
{
cout<< "This is base class\n";
}
};
class derived1 : public base
{
public:
void show()
{
cout<< "This is derived1 class.\n";
}
};
```

```
class derived2 : public base
{
public:
void show()
{
cout<< "This is derived2 class\n";
}
};
int main()
{
base b;
derived1 d1;
derived2 d2;
base *bptr;          // point to base

bptr = &b;
bptr->show();        // access base's show()

// point to derived1
bptr = &d1;
bptr->show();        // access derived1's show()

// point to derived2
bptr = &d2;
bptr->show();        // access derived2's show()

return 0;
}
```

OUTPUT:

```
This is base class
This is derived1 class
This is derived2 class
```

Explanation:

As the program illustrates, inside base, the virtual function show( ) is declared. Notice that the keyword virtual precedes the rest of the function declaration. When show( ) is redefined by derived1 and derived2, the keyword virtual is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, base is inherited by both **derived1** and **derived2**. Inside each class definition, show( ) is redefined relative to that class. Inside main( ), four variables are declared:

Name	Type

bptr — base class pointer

b — object of base

d1 — object of derived1

d2 — object of derived2

Next, bptr is assigned the address of b, and show( ) is called via bptr. Since bptr is pointing to an object of type base, that version of show( ) is executed. Next, bptr is set to the address of d1, and again show( ) is called by using bptr. This time bptr points to an object of type derived1. This causes derived1::show( ) to be executed. Finally, bptr is assigned the address of d2, and bptr->show( ) causes the version of show( ) redefined inside derived2 to be executed. The key point here is that the kind of object to which bptr points determines which version of show( ) is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

As another example,

Example 2:

```
#include<iostream>
using namespace std;
class Base
{
public:
virtual void show()
{
 cout<<" In Base \n";
}
};
class Derived: public Base
{
public:
void show()
{
  cout<<"In Derived \n";
}
};
int main()
{
Base *bp = new Derived;
bp->show();  // RUN-TIME POLYMORPHISM
return 0;
}
```

OUTPUT:

In Derived

The main thing to note about the above program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function.

1.1. PURE VIRTUAL FUNCTIONS

A Pure virtual function is a function which is declared in the base class, but has no definition relative to the base class. A function is made pure virtual by preceding its declaration with the keyword **virtual** and end it with **0**.

Therefore, this function in the base class is not used for any task. It's only a place holder. The General form of pure virtual function is:

virtual return_type function_name (arg_list) = 0;

Basic declaration of pure virtual function is:

```
Class SomeClass {
public:
virtual void pure_virtual() = 0; // a pure virtual function
  // note that there is no function body
};
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result. Hence definition for pure virtual function must be there in the derived class.

***Note:** If you do mistakenly try to give a declaration of a pure virtual function a definition as well, then the compiler will return an error when it comes across that code.*

When should pure virtual functions be used in C++?

In C++, a regular, "non-pure" virtual function provides a definition, which means that the class in which that virtual function is defined does not need to be declared abstract. You would want to create a pure virtual function when it doesn't make sense to provide a definition for a virtual function in the base class itself, within the context of inheritance.

For example, let's say that you have a base class called **Figure**. The Figure class has a function called draw(). And, other classes like **Circle** and **Square** derive from the Figure class. In the

Figure class, it doesn't make sense to actually provide a definition for the draw() function, because of the simple and obvious fact that a "Figure" has no specific shape. It is simply meant to act as a base class. Of course, in the Circle and Square classes it would be obvious what should happen in the draw() function ¨C they should just draw out either a Circle or Square (respectively) on the page. But, in the Figure class it makes no sense to provide a definition for the draw() function. And this is exactly when a pure virtual function should be used ¨C the draw() function in the Figure class should be a pure virtual function.

Example:

```
#include <iostream>
using namespace std;
class Base
{
public:
   virtual void show() = 0;  //Pure Virtual Function
};
class Derived1 :public Base
{
public:
        void show()
        {
        cout<< "In class Derived1"<<endl;
        }
};
class Derived2 :public Base
{
public:
        void show()
        {
        cout<< "In class Derived2"<<endl;
        }
};
int main()
{
Base *b;
Derived1 d1;
Derived2 d2;
```

```
b = &d1;
b->show();

b=&d2;
b->show();

return 0;
}
```

OUTPUT:

In Derived class1
In Derived class2

Note: *Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.*

c. Abstract Class:

A class that contains at least one pure virtual function is said to be abstract. We know that the specialty of a pure virtual function is to only declare the prototype of the function in a base class and not define it. i.e., the abstract class has, only the declaration of the function and the derived class takes care of the actual implementation.

Therefore, a class is known as an **abstract** class, if it contains at least one pure virtual function. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although we cannot create objects of an abstract class, we can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Thus an abstract class is a base class which acts a model for the derivation of other classes. Below program illustrates the usage of an abstract class.

Note: *For an abstract class in C++ there is no abstract keyword being used, since none exists in it.*

Example:

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

```
class Shape
{
        public:
        virtual void getdata() = 0;
        virtual float area() = 0;
        virtual float peri() = 0;
};
class square: public shape
{
        private:
        float side;
        public:
        voidgetdata()
        {
                cout<<"Type value of side";
                cin>>side;
        }
        float area()
        {
                return side*side;
        }
};
Class triangle : public shape
{
        private:
        floata,b,c;
        public:
        void getdata()
        {
                cout<<"Type value of a:";
                cin>>a;
                cout<<"Type value of b";
                cin>>b;
                cout<<"Type value of c";
                cin>>c;
        }
        float area()
        {
```

```
            float s = (a+b+c)/2.0;
            returnsqrt(s*(s-a)*(s-b)*(s-c));
        }
        float peri()
        {
            return (a+b+c);
        }
};
void main()
{
        square s;
        triangle t;
        cout<<"Square inputs:";
        s.getdata();
        cout<<"Area: "<<s.area();
        cout<<"Perimeter:"<<s.peri();
        cout<<"Triangle inputs:";
        t.getdata();
        cout<<"Area:"<<t.area();
        cout<<"Perimeter:"<<t.peri();

}
```

OUTPUT:

```
Square inputs:
        Type value of side: 5
        Area: 25
        Perimeter: 20
Triangle inputs:
        Type value of a: 3
        Type value of b: 4
        Type value of c: 5
        Area: 6
        Perimeter: 20
```

Why can't we create abstract class objects?

Abstract class has no implementation for method so; we can't create the object of an abstract class. If we try to create an **object** of the **abstract class** and then it calls the method having no body (as the method is pure virtual) compiler will give you an error.

1.2. EARLY VS. LATE BINDING

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding:

Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function. (Put differently, early binding means that an object and a function call are bound during compilation.)

binding is done during compilation.

For example:

```
#include <iostream>
void PrintValue(int nValue)
{
cout<<nValue;
}
int main()
{
PrintValue(5);          // This is a direct function call
return 0;
}
```

id → func call

Direct function calls can be resolved using a process known as early binding. Early binding (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember that all functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

eg:- normal fun^n calls.
overloading

Late binding:

The opposite of early binding is late binding. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is

eg:- virtual fun^n.

determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times. In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as late binding (or dynamic binding).

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the compiler can tell the CPU to jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

Chapter 2

Templates

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type and this is the main advantage.

2.1 GENERIC FUNCTIONS:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon.

For example, the Mergesort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different, only the resultant will be different. By creating a generic function, you can define the nature of the algorithm, independent of any data.

Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function.

In other words, A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

You can use templates to define functions as well as classes, let us see how do they work: The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
                    {
                       // body of function
                    }
```

Program to display largest among two numbers using function templates:

// If two characters are passed to function template, character with larger ASCII value is displayed.

```
#include <iostream>
using namespace std;
// template function
template <class T>
T Large(T n1, T n2)
{
        return (n1 > n2) ? n1 : n2;
}
int main()
{
        int i1, i2;
        float f1, f2;
        char c1, c2;
        cout << "Enter two integers:\n";
        cin >> i1 >> i2;
        cout << Large(i1, i2) <<" is larger." << endl;
        cout << "\nEnter two floating-point numbers:\n";
        cin >> f1 >> f2;
        cout << Large(f1, f2) <<" is larger." << endl;
        cout << "\nEnter two characters:\n";
        cin >> c1 >> c2;
        cout << Large(c1, c2) << " has larger ASCII value.";
        return 0;
}
```

Output:

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
```

z has larger ASCII value.

In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.

Large() function returns the largest among the two arguments using a simple conditional operation.

Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.

Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

As another example, We have swapping of two numbers of different data types.

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
        T temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}
int main()
{
        int i1 = 1, i2 = 2;
        float f1 = 1.1, f2 = 2.2;
        char c1 = 'a', c2 = 'b';
cout << "Before passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
Swap(i1, i2);
Swap(f1, f2);
Swap(c1, c2);
cout << "\n\nAfter passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
return 0;
}
```

Output:

```
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b
After passing data to function template.
```

```
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued. The Swap() function template takes two arguments and swaps them by reference.

a. A FUNCTION WITH TWO GENERIC TYPES:

You can define more than one generic data type in the template statement by using a comma-separated list. For Example,

```
#include <iostream>
using namespace std;
template <class type1, class type2> void myfun(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfun(10, "I like C++");
myfun(98.6, 19L);
return 0;
}
```

In this example, the placeholder types type1 and type2 are replaced by the compiler with the data types int and char *, and double and long, respectively, when the compiler generates the specific instances of myfunc( ) within main( ).

b. EXPLICITLY OVERLOADING A GENERIC FUNCTION:

Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called ***Explicit specialization***. If you overload a generic function, that overloaded function "hides" the generic function relative to that specific version. For Example,

```
#include <iostream>
#include <conio.h>
using namespace std;
template <class X> void swapargs(X &a, X
&b)
{
X temp;
temp = a;
a = b;
b = temp;
cout << "Inside template swapargs.\n";
}
// This overrides the generic version of
swapargs() for ints.
```

```
void swapargs(int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
cout << "Inside swapargs int specialization.\n";
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // calls explicitly overloaded swapargs()
swapargs(x, y); // calls generic swapargs()
swapargs(a, b); // calls generic swapargs()
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}
```

This Program Displays output as,

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

As the comments inside the program indicate, when **swapargs(i, j)** is called, it invokes the explicitly overloaded version of swapargs( ) defined in the program. Thus, the compiler does not generate this version of the generic swapargs( ) function, because the generic function is hidden by the explicit overloading.

Recently, a new-style syntax was introduced to denote the explicit specialization of a function. This new method uses the template keyword. For example, using the new-style specialization syntax, the overloaded swapargs( ) function from the preceding program looks like this.

```
// Use new-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
```

```
cout << "Inside swapargs int specialization.\n";
}
```

As you can see, the new-style syntax uses the **template<>** construct to indicate specialization. The type of data for which the specialization is being created is placed inside the angle brackets following the function name. This same syntax is used to specialize any type of generic function. While there is no advantage to using one specialization syntax over the other at this time, the new-style is probably a better approach for the long term.

c. OVERLOADING A FUNCTION TEMPLATE:

In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself. To do so, simply create another version of the template that differs from any others in its parameter list. For example:

```
// Overload a function template declaration.
#include <iostream>
using namespace std;
// First version of f() template.
template <class X> void f(X a)
{
cout << "Inside f(X a)\n";
}
// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
cout << "Inside f(X a, Y b)\n";
}
int main()
{
f(10); // calls f(X)
f(10, 20); // calls f(X, Y)
return 0;
}
```

Here, the template for f( ) is overloaded to accept either one or two parameters.

d. USING STANDARD PARAMETERS WITH TEMPLATE FUNCTIONS:

You can mix standard parameters with generic type parameters in a template function. These nongeneric parameters work just like they do with any other function. For Example:

```
// Using standard parameters in a template function.
```

```
#include <iostream>
using namespace std;
const int TABWIDTH = 8;
// Display data at specified tab position.
template<class X> void tabOut(X data, int tab)
{
for(; tab; tab--)
for(int i=0; i<TABWIDTH; i++) cout << ' ';
cout << data << "\n";
}
int main()
{
tabOut("This is a test", 0);
tabOut(100, 1);
tabOut('X', 2);
tabOut(10/3, 3);
return 0;
}
```

Here is the output produced by this program.

```
This is a test
        100
                X
                        3
```

In the program, the function tabOut( ) displays its first argument at the tab position requested by its second argument. Since the first argument is a generic type, tabOut( ) can be used to display any type of data. The tab parameter is a standard, call-by-value parameter. The mixing of generic and nongeneric parameters causes no trouble and is, indeed, both common and useful.

e. GENERIC FUNCTIONS RESTRICTIONS:

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions—only the type of data can differ.

Consider the overloaded functions in the following example program. These functions could not be replaced by a generic function because they do not do the same thing, in other words their

implementation part is different and in this case you can't create Template function even though their function name and number of parameters are same. For example,

```
#include <iostream>
#include <cmath>
using namespace std;
void myfunc(int i)
{
cout << "value is: " << i << "\n";
}
double myfunc(double d)
{
double multi;
multi=d*d;
return multi;
}
int main()
{
myfunc(1);
myfunc(12.2);
return 0;
}
```

OUTPUT:

value is: 1

2.2 GENERIC CLASS:

Like function templates, you can also create class templates for generic class operations. Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template <class Ttype>
```

```
class class-name {

.
.
}
```

Here, *Ttype* is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list.

How to create a class template object?

Once you have created a generic class, you create a specific instance of that class using the following general form:

class-name <type> ob;

Here, type is the type name of the data that the class will be operating upon. Member functions of a generic class are themselves automatically generic. You need not use template to explicitly specify them as such.

For example:

```
class-name<int> classObject;
class-name<float> classObject;
class-name<string> classObject;
```

Example:

Program to add, subtract, multiply and divide two numbers using class template

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private:
        T num1, num2;
public:
        Calculator(T n1, T n2)
        {
                num1 = n1;
                num2 = n2;
        }
        T add() { return num1 + num2; }
        T subtract() { return num1 - num2; }
        T multiply() { return num1 * num2; }
        T divide() { return num1 / num2; }
};
int main()
{
void displayResult()
{
cout << "Numbers are: " << num1 << " and "
<< num2 << "." << endl;
cout << "Addition is: " << add() << endl;
cout << "Subtraction is: " << subtract() <<
endl;
cout << "Product is: " << multiply() << endl;
cout << "Division is: " << divide() << endl;
}
```

```
{
        Calculator<int> intCalc(2, 1);
        Calculator<float> floatCalc(2.4, 1.2);
        cout << "Int results:" << endl;
        intCalc.displayResult();
cout << endl << "Float results:" << endl;
        floatCalc.displayResult();
return 0; }
```

Output:

int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2

a. An Example with Two Generic Data Types:

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the template specification.

```
#include <iostream>
using namespace std;
template <class Type1, class Type2>
class myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates add power.");
ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}
```

Output:

10 0.23
X Templates add power.

b. Using Non-Type Arguments with Generic Classes:

In the template specification for a generic class, you may also specify non-type arguments. That is, in a template specification you can specify what you would normally think of as a standard argument, such as an integer or a pointer. The syntax to accomplish this is essentially the same as for normal function parameters: simply include the type and name of the argument. General Syntax,

```
template <class TType, int arg> class class_name {
...
};
```

c. Explicit Class Specializations

As with template functions, you can create an explicit specialization of a generic class. To do so, use the **template<>** construct, which works the same as it does for explicit function specializations. For example:

```
// Demonstrate class specialization.
#include <iostream>
using namespace std;
template <class T> class myclass {
T x;
public:
myclass(T a) {
};
cout << "Inside generic myclass\n";
x = a;
}
T getx() { return x; }
// Explicit specialization for int.
template <> class myclass<int> {
int x;
public:
};
myclass(int a) {
cout << "Inside myclass<int> specialization\n";
```

```
x = a * a;
}
int getx() { return x; }
int main()
{
 myclass<double> d(10.1);
 cout << "double: " << d.getx() << "\n\n";
 myclass<int> i(5);
 cout << "int: " << i.getx() << "\n";
 return 0;
 }
```

This program displays the following output:

```
Inside generic myclass
double: 10.1
Inside myclass<int> specialization
int: 25
```

In the program, pay close attention to this line:

```
template <> class myclass<int> {
```

It tells the compiler that an explicit integer specialization of **myclass** is being created. This same general syntax is used for any type of class specialization. Explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processed by the compiler.

Chapter 3

Exception Handling

I. Exception Handling Fundamentals:

Many times while executing programs we come across multiple types of ~~errors~~ or ~~disrupts~~, which may lead to termination of your program. To continue execution of the program even after the occurrence of errors can be achieved through some handling mechanisms, this is in other terms called as *Exception handling*.

An exception is a problem/error that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, this infact is the advantage of C++ over C platform.

Such exceptions can be classified as an attempt to divide by zero, some memory related exceptions like array out of bound etc.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,**and **throw**.

- **try:** Program statements that you want to monitor for exceptions are contained in a try block. A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- **throw:** A program throws an exception when a problem shows up in try block. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem in a particular manner. The **catch** keyword indicates the catching of an exception.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
        //try block
}
catch( Type e)
```

```
{
  // catch block
}
```

Throw *Exception*;

The try can be as short as a few statements within one function or as all-encompassing as enclosing the main( ) function code within a try block.

When an exception is thrown, it is caught by its corresponding catch statement, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception. That is, if the data type specified by a catch matches that of the exception, then that catch statement is executed (and all others are bypassed).

When an exception is caught, arg will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the try block), then no catch statement is executed.

throw generates the exception specified by exception. If this exception is to be caught, then throw must be executed either from within a try block itself, or from any function called from within the try block (directly or indirectly).

If you throw an exception for which there is no applicable catch statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function terminate( ) to be invoked. By default, terminate( ) calls abort( ) to stop your program.

Following are main advantages of exception handling over traditional error handling:

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Example:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

This program displays the following output:

Start
Inside try block
Caught an exception -- value is: 100
End

Look carefully at this program. As you can see, there is a try block containing three statements and a catch(int i) statement that processes an integer exception. Within the try block, only two of the three statements will execute: the first cout statement and the throw.

Once an exception has been thrown, control passes to the catch expression and the try block is terminated. That is, catch is not called. Rather, program execution is transferred to it. Thus, the cout statement following the throw will never execute.

Usually, the code within a catch statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the catch.

However, often an error cannot be fixed and a catch block will terminate the program with a call to exit( ) or abort( ).

a. Catching Class type:

An exception can be of any type,built-in or non-built-in including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following is the syntax:

```
//Normal syntax
try
{
  Throw Exception;
}
catch( type e1 )
{
  // catch block
}
// class type
Class ExceptionDemo
{}
Try
{
        Throw ExceptionDemo;
}
Catch(ExceptionDemo ex)
{
  //catch block
}
```

Example:

```
int main()
{
int i;
try {
cout << "Enter a positive number: ";
cin >> i;
if(i<0)
throw MyException("Not Positive", i);
```

```
}
catch (MyException e) { // catch an error
cout << e.str_what << ": ";
cout << e.what << "\n";
}
return 0;
}
```

Output:

```
Enter a positive number: -4
Not Positive: -4
```

The program prompts the user for a positive number. If a negative number is entered, an object of the class MyException is created that describes the error. Thus, MyException encapsulates information about the error. This information is then used by the exception handler.

b. Using Multiple Catch Statements:

As stated, you can have more than one catch associated with a try. In fact, it is common to do so. However, each catch must catch a different type of exception. General syntax:

```
Try
{
}
Catch(exception1 e1)
{
}
Catch(exception2 e2)
{
}
Catch(exception3 e3)
{
}
```

Example:

```
void Xhandler(int test)
{
Try
{
if(test) throw test;
else throw "Value is zero";
}
catch(int i)
```

```
{
cout << "Caught Exception #: " << i << '\n';
}
catch(const char *str)
{
cout << "Caught a string: ";
cout << str << '\n';
}
}
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}
```

Output:

Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End

As you can see, each catch statement responds only to its own type. In general, catch expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other catch blocks are ignored.

II. Handling Derived Class Exception:

You need to be careful how you order your catch statements when trying to catch exception types that involve base and derived classes because a catch clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the catch sequence. If you don't do this, the base class catch will also catch all derived classes. For example, consider the following program.

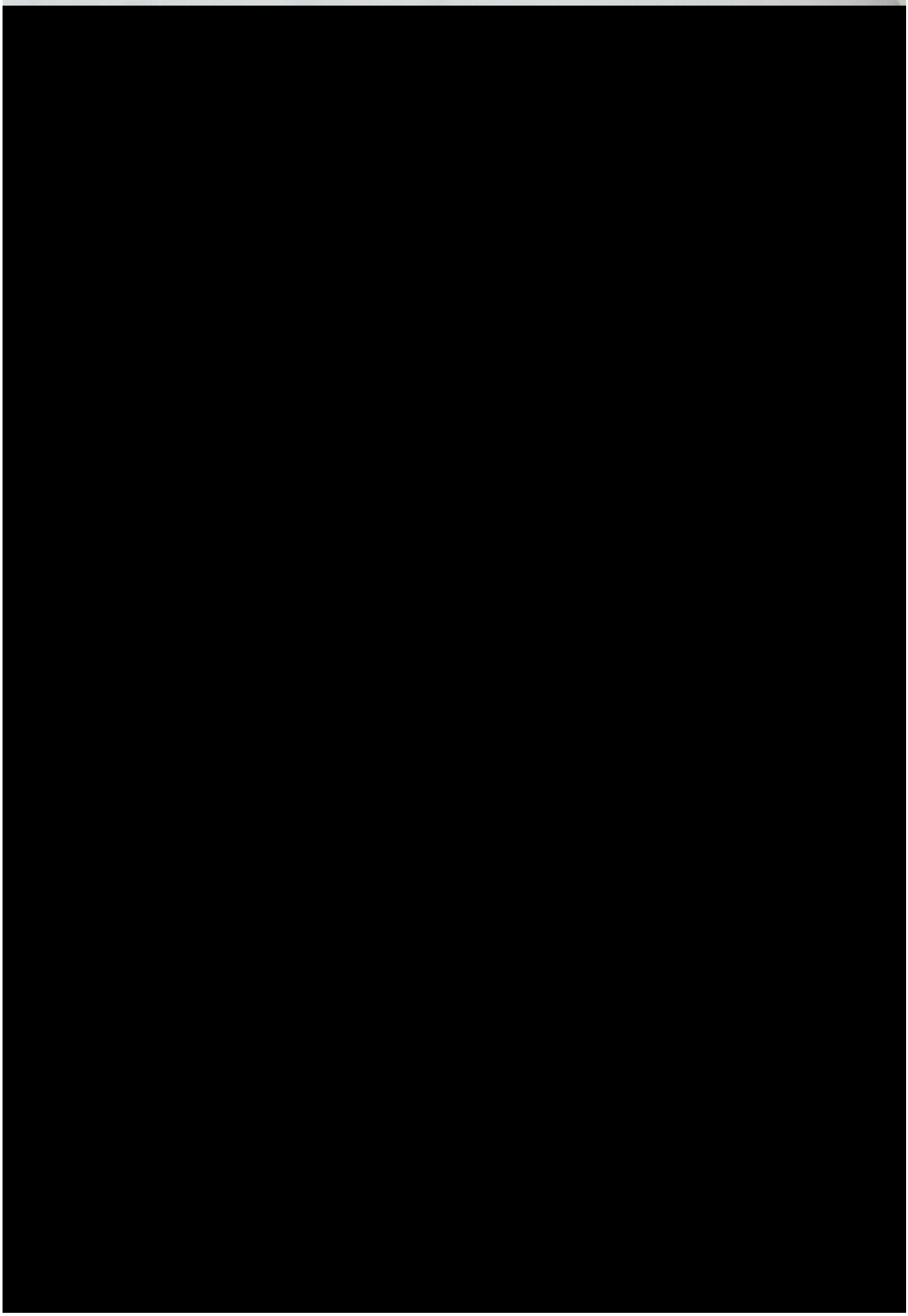
```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main()
{
D derived;
try {
throw derived;
}
catch(B b) {
cout << "Caught a base class.\n";
}
catch(D d) {
cout << "This won't execute.\n";
}
return 0;
}
```

Here, because derived is an object that has B as a base class, it will be caught by the first catch clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the catch clauses.

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached.

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modifed program and it prints “Caught Derived Exception”

```
#include<iostream>
using namespace std;
class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  // some other stuff
```



```
}
Catch(...) { // catch all exceptions
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```

As you can see, all three throws were caught using the one catch statement.

b. Restricting Exceptions:

You can restrict the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a throw clause to a function definition.

The general form of this is shown here:

```
ret-type func-name(arg-list) throw(type-list)
{
// ...
}
```

Here, only those data types contained in the comma-separated type-list may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw any exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function unexpected( ) to be called. By default, this causes abort( ) to be called, which causes abnormal program termination.

The following program shows how to restrict the types of exceptions that can be thrown from a function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
int main()
{
cout << "start\n";
try{
Xhandler(0); // also, try passing 1 and 2 to Xhandler()
}
catch(int i) {
cout << "Caught an integer\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double d) {
cout << "Caught double\n";
}
cout << "end";
return 0;
}
```

In this program, the function Xhandler( ) may only throw integer, character, and double exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, unexpected( ) will be called.)

To see an example of this, remove int from the list and retry the program. It is important to understand that a function can be restricted only in what types of exceptions it throws back to the try block that called it. That is, a try block within a function may throw any type of exception so long as it is caught within that function. The restriction applies only when throwing an exception outside of the function.

c. Rethrowing an Exception:

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence. An exception can only be rethrown from within a catch block. When an exception is rethrown, it is propagated outward to the next catch block.

The most likely reason for doing so is to allow multiple handlers access to the exception. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement. The following program illustrates rethrowing an exception, in this case a char * exception. For example,

```
#include <iostream>
using namespace std;
void myFunction()
{
 try
 {
  throw "hello";   // throw a char *
 }
 catch(const char *)
 {                  // catch a char *
  cout << "Caught char * inside myFunction\n";
  throw ;          // rethrow char * out of function
 }                 // myFunction
}
int main()
{
 cout << "Start\n";
 try
 {
  myFunction();
 }
 catch(const char *)
 {
  cout << "Caught char * inside main\n";
 }
 cout << "End";
 return 0;
```

}

Program Displays,

Start

Caught char * inside myFunction

Caught char * inside main

End