Unit – 5 virtual function and polymorphism

**Pointers :** - it one of the key aspect of C++ language similar to that of C. pointers are unique approach to handle data in C and c++.

A pointer is a derived data type that refers to another data variable by storing the variable’s memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer can also refer to another pointer in c++.

**Address :-** the key concept is in computer, every byte in the computer’s memory has an address. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there 1,2, 3, and so on. If we have 640 KB memory , the highest address is

655359.

Our program, when it is loaded onto memory; occupies a certain range of these address. That means that every variable and every function in your program starts at a particular address.

**The address of operator &**

We can find out the address occupied by a variable by using the address of operator &. For example:

this simple progra declare three variable a, b, and c. its initial value are

10, 20 and 30, and their address are printed.

#include<iostream.h>

#include<conio.h>

void main()

{

int a=10,b=20,c=30;

cout<<"\n"<<&a;

cout<<"\n"<<&b;

cout<<"\n"<<&c;

getch();

}

**Pointer variable :** address by themselves are rather limited. It is nice to know that we can find out where things are in memory. As we did in VARADDR, but printing out address values is not all that useful. The potential for increasing our programming power requires an additional idea: *variables that hold address values.* **A variable that holds the address of a another variable is called pointer variable.**

**For example**

#include<iostream.h>

#include<conio.h>

void main()

{

int a=10,b=20;

int \*ptr;

clrscr();

cout<<"\n address of a is "<<&a;

cout<<"\n address of b is "<<&b;

ptr=&a;

cout<<"\n value of a pinter is "<<ptr;

cout<<"\n value of A through pointer is "<<\*ptr;

ptr=&b;

cout<<"\n value of pinter is "<<ptr;

cout<<"\n value of B through pointer is "<<\*ptr;

getch();

}

**Pointer and Arrays :-** there is a close association between pointers and arrays. The array can be easily accessed by using the pointer.

For example:

WAP to initialized these numbers into an array(10,20,3,40,50) and display using pointer.

void main()

{

int num[5]={10,20,30,40,50};

int i, \*ptr;

ptr=&num[0];

for(i=0;i<5;i++)

{

cout<<"\n"<<\*(ptr+i);

}

getch();

}

**Memory Management : *new*** AND ***delete***

**We have** seen many example of array. To declare an array , it means we have to fixed to array size for example:

Int num[20];

It means we declared an array **num** and its size is twenty elements, to store 20 numbers, and not more and but can be less than 20. it is called static array declarations, we the size of array can not be increase or decrease at the time of program is running. But there is great problem , that at the beginning of program , we can not expect exact the number to used or entered. So for the solution of this problem is **new .** the new operator is used to allocate the memory run time. By using the **new** operator to allocate the block of memory while the program is running. For example:

#include<iostream.h>

#include<conio.h>

void main()

{

int \*num;

int size,i;

cout<<"enter the size of array";

cin>>size;

num=new int[size]; //allocating memory run time.

cout<<"enter number"<<size;

for(i=0;i<size;i++)

{

cin>>num[i];

}

for(i=0;i<size;i++)

cout<<num[i];

getch();

}

**Delete operator:-** if our program reserves many chunks of memory using **new. Eventually,** all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system.

delete ptr;

#include<iostream.h>

#include<conio.h>

void main()

{

int \*num;

int size,i;

cout<<"enter the size of array";

cin>>size;

num=new int[size]; //allocating memory run time.

cout<<"enter number"<<size;

for(i=0;i<size;i++)

{

cin>>num[i];

}

for(i=0;i<size;i++)

cout<<num[i];

**delete num; //releasing the memory created by new operator.**

getch();

}

**The *this* pointer :**

The member functions of every object have access to a sort of magic named **this** , which point to the object itself. Thus any member function can find out the address of the object of which it is a member.

When we call a member function, it comes into existence with the value of this set to the address of the object for which it was called. The this pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to for example:

#include<iostream.h>

#include<conio.h>

class tthis

{

public:

int a,b;

void get(int a, int b)

{

**this->a=a;**

**this->b=b;**

}

void disp()

{

cout<<a;

cout<<b;

}

};

void main()

{

tthis obj;

obj.get(100,200);

obj.disp();

getch();

}

**Introduction :-** polymorphism is one of the crucial features of OOP. It simply means one name , multiple forms. The concept of polymorphism is implemented using the overloaded functions and operators. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called **early binding or static binding or static linking.**

**It would** be nice if the appropriate member function could be selected while the program is running. This is known as ***run time polymorphism***. C++ supports a mechanism knows as **virtual function** to achieve run time polymorphism.

Polymorphism

Function over loading

Operator overloading

Virtual function

Achieving polymorphism

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked . since the function is linked with a particular class much later after the compilation , this process is termed as **late binding or dynamic binding.** Because selection of the appropriate function is done dynamically at run time.

**Virtual functions** : polymorphism refers to the to the property by which objects belonging to different classes are able to respond to the same message but in different forms. An essential requirements of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. As we discovered that a base pointer , even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of pointer . how do we then achieve polymorphism ? it is achieved using what is known as “virtual function”

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration. When a function is made **virtual, C++** determines which function to use at run time based on the type of object pointed by the base pointer, rather than the type of pointer. Thus by making the base pointer to point to different objects, we can execute different versions of the **virtual function.**  It is also called dynamic binding or late binding.

Example is :

#include<iostream.h>

#include<conio.h>

class base

{

public:

void disp()

{

cout<<"\n this is base class";

}

virtual void show()

{

cout<<"\n this is base class virtual function";

}

};

class child :public base

{

public:

void disp()

{

cout<<"\n \n this derived class fucntion";

}

void show()

{

cout<<"\n this is derived class virtual fuction";

}

};

void main()

{

base \*pobj;

base bobj;

child cobj;

pobj=&bobj;

pobj->disp(); //calls base class function

pobj->show(); //calls base class function

pobj=&cobj;

pobj->disp(); //calls base class function

pobj->show(); //calls child class function

getch();

}

**Rules for virtual functions:**

1. the virtual function must be members of some class.
2. they can not be static members.
3. they are accessed by using object pointers.
4. a virtual function can be friend of another class.
5. we can not have virtual constructor.
6. while a base pointer can point to any type of the derived object, the reverse is not true.
7. when a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class.
8. if a virtual function defined in base class , no need to again defined in the derived class.

**Pure virtual functions :**

**It is** normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder.*  Such function are called *do-nothing functions.*

**Do** nothing function may be define as.

Virtual type functionName()=0;

Such function are called *pure virtual function.* Pure virtual function declared in base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. **Remember that a class containing pure virtual functions can not be used to declare any object of its own.** Such class are called *abstract class.* The main objective of an abstract class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Abstract class

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

For example:

#include<iostream.h>

#include<conio.h>

class abstract

{

public:

virtual void display()=0; //pure virtual function

virtual void get(int,int)=0; //pure virtual function

};

class derived : public abstract

{

public:

int a, b;

void get(int x,int y)

{

a=x;

b=y;

}

void display()

{

int sum=0;

sum=a+b;

cout<<"\n sum of two number is "<<sum;

}

};

void main()

{

derived ob;

ob.get(100,200);

ob.display();

getch();

}

**Exception Handling**

Compile time exception handling, Run Time exception handling

we know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are *logic errors and syntactic errors.* The logic errors occurred due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these types of errors, exhaustive debugging and testing procedures.

We often come across some peculiar problems other than logic or syntax errors. They are known s *exceptions.* Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

**Basic of exception handling**

Exception are of two kinds namely, *synchronous exceptions* and *asynchronous exceptions.* Errors such as “out-of-range index” and “over-flow” belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts. Memory insufficient, processor errors) are called asynchronous exceptions. So, in c++ it is possible to handle only synchronous exceptions only.

The process of the exception handling mechanism is to provide means to detect and report an “exceptional circumstance” so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks.

1. find the problem(hit the exception)
2. inform that an error has occurred(throw the exception)
3. receive the error information(catch the exception)
4. take corrective actions(handle the errors)

**Exception Handling Mechanism :-** C++ exception handling mechanism is basically built upon three keywords, namely, **try, throw, and catch.** The keyword **try,** is used to preface a block of statements which may generate exceptions. This block of statements is known as *try block* . when an exception is detected , it is thrown using a ***throw*** statement in the try block. The *catch* block , and handled it appropriately.

The **catch** block that catches an exception must immediately follow the **try** block that throws the exception . the general form of these two blocks are as follows:

………..

………….

Try

{

…………..

……………

**Throw** exception;

**………….**

………….

}

Catch(type arg)

{

……………..

………………

…………….

Handle the exception

}

…………

………….

#include<iostream.h>

#include<conio.h>

void main()

{

int a,b,c;

cout<<"enter two numbers";

cin>>a>>b;

try

{

if (b==0)

{

throw b;

}

else

{

c=a/b;

cout<<"division result is "<<c;

}

}

catch(int b)

{

cout<<"value of denominator is zero "<<b;

}

cout<<"\n the prom is over";

getch();

}

The general form of code for function type of program.

Type function(list)

{

…….

………

Throw object;

……..

………

}

Catch(type arg)

{

……..

……….

……….

}

………

………

For example

Demonstrates how a try block invokes a function that generate an exception.

#include<iostream.h>

#include<conio.h>

void get(int a,int b)

{

int div=0;

if(b==0)

{

throw b;

}

else

{

div=a/b;

cout<<"\n division is "<<div;

}

}

void main()

{

int a,b;

cout<<"enter two numbers";

cin>>a>>b;

try

{

get(a,b);

}

catch(int b)

{

cout<<"value of divider is "<<b;

}

cout<<"\n the prom is over";

getch();

}

**Throwing mechanism :-** when an exception that is desired to be handled is detected. It is thrown using the *throw* statement in one of the following forms.

Throw (exception);

Throw exception;

The exception may be any type object , including constant.

When an exceptin is thrown, it will be caught by the **catch** statement associated with the **try** block. That is the control exits the current **try block,** and is transferred to the catch block after that **try** block.

**Catching mechanism :** code for handling exception is included in catch blocks. A catch block looks like a function definition and is of the form

Catch(type arg)

{

//statements for managing exception

}

The *type* indicates the type of exception that catch block handles. The parameter arg is an optional parameter name.

**Multiple catch statements :**

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try block like as

Try

{

………

………..

……….

}

Catch(type1 arg)

{

……….

………..

}

Catch(type2 arg)

{

………..

…………

}

Catch(typeN arg)

{

………..

…………

}

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed . after executing the handler , the control goes to the first statement after the last **catch** block for that **try**.

**For example:**

**#include<iostream.h>**

**#include<conio.h>**

**main()**

**{**

**int a,b,c;**

**float x,y,z;**

**cout<<"enter two numbers";**

**cin>>a>>b;**

**cout<<"\n enter two floating point number ";**

**cin>>x>>y;**

**try**

**{**

**if (b==0)**

**{**

**throw b;**

**}**

**else**

**{**

**c=a/b;**

**cout<<"division result is "<<c;**

**}**

**if(y==0)**

**throw y;**

**else**

**z=x/y;**

**cout<<"\n floating division is "<<z;**

**}**

**catch(int b)**

**{**

**cout<<"int value of denominator is zero "<<b;**

**}**

**catch(float y)**

**{**

**cout<<"\n floating value is zero";**

**}**

**cout<<"\n the programer is over";**

**getch();**

**}**