**Program No:** 01

**Program Topic:** *Pointers to Derived Class*

**Program Title:**

Write a C++ program to access a derived class by using a base class pointer.

**Objective:**

To learn how to access derived class members which are inherited by base class using a base class pointer.

**Syntax:**

base \*p;

base base\_ob;

derived derived\_ob;

p = & base\_ob;

p = & derived\_ob;

**Source Code:**

#include <iostream>

using namespace std;

class base{

int x;

public :

void setx(int i){x = i;}

int getx(){return x;}

};

class derived : public base{

int y;

public :

void sety(int i){y = i;}

int gety(){return y;}

};

int main()

{

base \*p;

base b\_ob;

derived d\_ob;

p = &b\_ob;

p->setx (30);

cout << "Base object x: " << p->getx () << endl;

p = &d\_ob;

p->setx (33);

d\_ob.sety (77);

cout << "Derived object x: " << p->getx() << endl;

cout << "Derived object y: " << d\_ob.gety() << endl;

return 0;

}

**Output:**

Base object x: 30

Derived object x: 33

Derived object y: 77

Process returned 0 (0x0) execution time : 0.107 s

Press any key to continue.

**Explanation:**

A base pointer can point to an object of any class derived from that base without generating a type mismatch error. Although a base pointer can be used to point to a derived object, it can access only those members of the derived object that are inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the members added by the derived class. But a pointer of the derived type cannot be used to access an object of the base class.

**Program No:** 02

**Program Topic:** *Virtual Function*

**Program Title:**

Write a C++ program using a virtual function.

**Objective:**

To learn about virtual functions.

**Syntax:**

virtual data\_type func\_name(parameter\_list)

{

//… … …

}

**Source Code:**

#include <iostream>

using namespace std;

class base

{

public:

int i;

base (int x) {i = x;}

virtual void func(){

cout << "Using base version of func (): ";

cout << i << endl;

}

};

class derived1 : public base

{

public:

derived1(int x) : base (x) {}

void func(){

cout << "Using derived1 ’s version of func (): ";

cout << i\*i << endl;}

};

class derived2 : public base

{

public :

derived2(int x) : base (x) {}

void func(){

cout << "Using derived2 ’s version of func (): ";

cout << i+i << endl;

}

};

int main ()

{

base \*p;

base ob (10);

derived1 d\_ob1 (10);

derived2 d\_ob2 (10);

p = &ob;

p-> func ();

p = & d\_ob1;

p-> func ();

p = & d\_ob2;

p-> func ();

return 0;

}

**Output:**

Using base version of func (): 10

Using derived1 Æs version of func (): 100

Using derived2 Æs version of func (): 20

Process returned 0 (0x0) execution time : 0.552 s

Press any key to continue.

**Explanation:**

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, it needs to be preceded the function’s declaration with the keyword virtual. A virtual function can be called just like any other member function. However, what makes a virtual function interesting-and capable of supporting run-time polymorphism-is what happens when a virtual function is called through a pointer. When a base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer. And, this determination is made at run time. The redefinition of a virtual function inside a derived class might, at first, seem somewhat similar to function overloading. However, the two processes are distinctly different. First, an overloaded function must differ in type and/or number of parameters, while a redefined virtual function must have precisely the same type and number of parameters and the same return type.

**Program No:** 03

**Program Topic:** *Pure Virtual Function*

**Program Title:**

Write a C++ program to determine areas of rectangle and triangle using pure virtual function.

**Objective:**

To learn about pure virtual function.

**Syntax:**

virtual type func\_name ( parameter\_list ) = 0;

**Source Code:**

#include <iostream>

using namespace std;

class area

{

double dim1 , dim2;

public :

void setarea(double d1 , double d2){

dim1 = d1;

dim2 = d2;

}

void getdim(double &d1 , double &d2){

d1 = dim1;

d2 = dim2 ;

}

virtual double getarea () = 0;

};

class rectangle : public area{

public :

double getarea (){

double d1 , d2;

getdim (d1 , d2);

return d1 \* d2;

}

};

class triangle : public area{

public :

double getarea (){

double d1 , d2;

getdim (d1 , d2);

return 0.5 \* d1 \* d2;

}

};

int main ()

{

area \*p;

rectangle r;

triangle t;

r. setarea (3.3 , 7.7) ;

t. setarea (4.0 , 8.0) ;

p = &r;

cout << "Rectangle has area : " << p-> getarea () << endl;

p = &t;

cout << "Triangle has area : " << p-> getarea () << endl;

return 0;

}

**Output:**

Rectangle has area : 25.41

Triangle has area : 16

Process returned 0 (0x0) execution time : 0.123 s

Press any key to continue.

**Explanation:**

A pure virtual function has no definition relative to the base class. Only the function’s prototype is included. The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guarantee that a derived class will provide its own redefinition. When a class contains at least one pure virtual function, it I referred to as an abstract class. Since an abstract class contains at least one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited.

**Program No:** 04

**Program Topic:** *Generic Function*

**Program Title:**

Write a C++ program to swap the value of two different types variable using generic functions.

**Objectives:**

To learn about generic functions.

**Syntax:**

template < class Ttype > return\_type func\_name ( parameter list )

{

// body of function

}

**Source Code:**

#include <iostream>

using namespace std ;

template <class X> void swapargs (X &a, X &b)

{

X temp ;

temp = a;

a = b;

b= temp ;

}

int main ()

{

int i = 1, j = 2;

float x = 1, y = 3.3;

cout << "Original i, j: " << i << ' ' << j << endl ;

cout << "Original x, y: " << x << ' ' << y << endl ;

swapargs (i, j);

swapargs (x, y);

cout << "Swapped i, j: " << i << ' ' << j << endl ;

cout << "Swapped x, y: " << x << ' ' << y << endl ;

return 0;

}

**Output:**

Original i, j: 1 2

Original x, y: 1 3.3

Swapped i, j: 2 1

Swapped x, y: 3.3 1

Process returned 0 (0x0) execution time : 0.042 s

Press any key to continue.

**Explanation:**

A generic function defines a general set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data. A generic function is created using the keyword template. The normal meaning of the word template accurately reflects the keyword’s use in C++. It is used to create a template that describes what a function will do, leaving it to the compiler to fill in the details as needed.

**Program No:** 05

**Program Topic:** *Overriding Template Function.*

**Program Title:**

Write a C++ program to override a template function.

**Objective:**

To learn more about template function that it can be overridden too.

**Syntax:**

template < class Ttype > return\_type func\_name ( parameter list )

{

// body of function

}

return\_type func\_name (parameter\_list)

{

//… …. …

}

**Source Code:**

# include <iostream>

using namespace std ;

template <class X> void swapargs (X &a, X &b)

{

X temp ;

temp = a;

a = b;

b= temp ;

}

void swapargs ( int a, int b)

{

cout << "Inside swapargs(int ,int )\n";

}

int main ()

{

int i=10 , j =20;

float x=10 , y =23.3;

cout << "Original i, j: " << i << ' ' << j << endl ;

cout << "Original x, y: " << x << ' ' << y << endl ;

swapargs (i, j); // calls overloaded swapargs ()

swapargs (x, y); // swap floats

cout << "Swapped i, j: " << i << ' ' << j << endl ;

cout << "Swapped x, y: " << x << ' ' << y << endl ;

return 0;

}

**Output:**

Original i, j: 10 20

Original x, y: 10 23.3

Inside swapargs(int ,int )

Swapped i, j: 10 20

Swapped x, y: 23.3 10

Process returned 0 (0x0) execution time : 0.033 s

Press any key to continue.

**Explanation:**

Even though a template function overloads itself as needed, you can explicitly overload one, too. If one generic function is overloaded, that overloaded function overrides (or hides) the generic function relative to that specific version.

**Program No:** 06

**Program Topic:** *Generic Class*

**Program Title:**

Write a C++ program using generic classes.

**Objective:**

To learn generic class.

**Syntax:**

template <class Ttype > class class\_name

{

//. . .

};

… … …

class\_name <type > ob;

**Source Code:**

#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 (1 , 0.23) ;

myclass <char, char \*> ob2('X', " This is a test ");

ob1 . show ();

ob2 . show ();

return 0;

}

**Output:**

1 0.23

X This is a test

Process returned 0 (0x0) execution time : 0.031 s

Press any key to continue.

**Explanation:**

In addition to defining generic functions, generic classes also can be defined. When it is done, a class is created that defines all algorithms used by that class, but the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.