Inheritance Example:

/\* Single Inheritance in C++:

When the derived class inherits only one base class,

it is known as Single Inheritance.

\*/

#include <iostream>

using namespace std;

class Base

{public:

double salary = 875;

};

//============================================

class Derived : public Base

{public:

double bonus = 87.50;

void sum() {

cout << "Your Total Salary is: " << (salary + bonus) << endl;

}

};

//==========================================================

int main()

{ // Creating an object of the derived class.

Derived x;

// Gets the salary variable of Base class.

cout << "Your Salary is:" << x.salary << endl;

// Gets the bonus variable of the Derived class.

cout << "Your Bonus is:" << x.bonus << endl;

x.sum();

return 0;

}

/\* Multiple Inheritance in C++

When a derived class(child class) inherits more than

one base class(parent class), it is called multiple inheritance.

\*/

#include <iostream>

using namespace std;

class Base1

{public: float salary = 1500;

};

class Base2

{public:

float bonus = 120;

};

class Derived : public Base1, public Base2

{public:

void sum()

{

cout << "Your Total Salary is: " << (salary + bonus) << endl;

}

};

//=================================================================

int main()

{ // Creating an object of the derived class and calling the sum() function.

Derived x;

x.sum();

return 0;

}

/\* Multilevel Inheritance in C++:

When a derived(child) class inherits the base class

and acts as the base class(parent class) to the other class,

it is called Multilevel Inheritance.

There can be any number of levels i.e any number of derived

classes in multilevel inheritance.

\*/

#include <iostream>

using namespace std;

class BaseClass

{public:

void print()

{

cout << "This is an example of Multilevel Inheritance";

}

};

class DerivedClass1 : public BaseClass {};

//============ Here class is derived from another derived class.

class DerivedClass2 : public DerivedClass1 {};

//==============================================================

int main()

{

DerivedClass2 Obj;

Obj.print();

return 0;

}

/\* Hierarchical Inheritance in C++:

When more than one class is inherited from a single base class,

it is called Hierarchical Inheritance.

\*/

#include <iostream>

using namespace std;

class Single\_base\_class

{public: int x, y;

void data()

{ cout << "\nEnter the value of x and y ->\n";

cin >> x >> y;

}

};

// Class derived1 is derived from single\_base\_class.

class Derived1 : public Single\_base\_class

{public: void product()

{ cout << "\nThe Product is= " << x \* y; }

};

// Class derived1 is also derived from class single\_base\_class.

class Derived2 : public Single\_base\_class

{public: void sum()

{ cout << "\nThe Sum is= " << x + y; }

};

//===============================================================

int main()

{ // Creating objects of both classes and calling functions on them.

Derived1 obj1;

Derived2 obj2;

obj1.data();

obj1.product();

obj2.data();

obj2.sum();

return 0;

}

/\* Hybrid Inheritance in C++:

It is a combination of one or more types of inheritance.

Now that you practice Inheritance we can say :

Advantages of Inheritance in C++ are:

- Inheritance in C++ promotes Code reusability.

When a derived class inherits the base class,

then the derived class can access all the functionality,

and the base class's code can be reused in the derived class.

- It improves code readability as you don’t have to rewrite the

same code repeatedly; hence, the code looks cleaner and readable.

- It saves time and effort as the code is already written and is

inherited; therefore, it saves time to write code again.

- Inheritance supports extensibility as new classes can be easily

added to existing classes.

\*/

#include <iostream>

using namespace std;

class World

{public: World()

{ cout << "This is World!\n"; }

};

//===================== Here is Single Inheritance ==================.

class Continent : public World

{public: Continent()

{ cout << "This is Continent\n"; }

};

//===================== Country Class =================================

class Country

{ public: Country()

{ cout << "This is the Country\n"; }

};

//======================== Here is multiple Inheritance=====================

class UnitedState : public Continent, public Country

{public: UnitedState()

{ cout << "This is UnitedState!"; }

};

//=============================================================================

int main()

{

UnitedState myworld;

return 0;

}

Polymorphism 1

/\*

When the same entity (function or object)

behaves differently in different scenarios,

it is known as Polymorphism in C++

Consider addition operation :

The “ +” operator in c++ can perform two specific functions

at two different scenarios. When the “+” operator is used

in numbers, it performs addition.

int a = 7;

int b = 3;

int sum = 7 + 3; // sum =10

And the same “+” operator is used in the string, it performs

concatenation.

string Fname = "Ali ";

string Lname = "Nikzad";

string FLNAME = Fname + Lname; // FLNAME = "Ali Nikzad"

1. Compile Time Polymorphism

In compile-time polymorphism, a function is called at the

time of program compilation. We call this type of polymorphism

as early binding or Static binding.

Function overloading and operator overloading is the type of

Compile time polymorphism.

below example is FUNCTION OVERLOADING

\*/

#include <iostream>

#include <string>

using namespace std;

class Addition

{ public: int ADD(int X, int Y) // Function with parameter

{

return X + Y; // this function is performing addition of two Integer value

}

string ADD()

{ // Function with same name but without parameter

string a = "HELLO";

string b = " Everyone"; // in this function concatenation is performed

return a + b;

}

};

int main(void)

{

Addition obj; // Object is created

cout << obj.ADD(20, 30) << endl; //first method is called

cout << obj.ADD()<<endl; // second method is called

return 0;

}

/\*

Next type of Polymorphism is example of

II. Operator Overloading

In Operator overloading means defining additional

tasks to operators without changing its actual meaning.

We do this by using operator function.

The purpose of operator overloading is to provide a special

meaning to the user-defined data types.

The advantage of Operators overloading is to perform different

operations on the same operand.

\*/

#include <iostream>

using namespace std;

class A

{private: string x;

public: A() {}

A(string i)

{ x = i; }

void operator+(A);

};

//================================

void A:: operator+(A a)

{ string m = x + a.x;

cout << "The result of the addition of two objects is : " << m << endl;

}

int main()

{

A a1("Welcome");

A a2(" Everyone");

a1 + a2;

return 0;

}

/\*2. Runtime Polymorphism

In a Runtime polymorphism, functions are called at

the time the program execution. Hence, it is known as

late binding or dynamic binding.

A. Function overriding is a part of runtime polymorphism.

In function overriding, more than one method has the

same name with different types of the parameter list.

It is achieved by using virtual functions and pointers.

It provides slow execution as it is known at the run time.

Thus, It is more flexible as all the things executed at the run time.

A. Function overriding

In function overriding, we give the new definition to base class

function in the derived class. At that time, we can say the base

function has been overridden. It can be only possible in the

‘derived class’. In function overriding, we have two definitions

of the same function, one in the superclass and one in the derived

class. The decision about which function definition requires calling

happens at runtime. That is the reason we call it ‘Runtime polymorphism’.

\*/

#include <iostream>

using namespace std;

class Animal

{public: void function()

{ cout << "Eating..." << endl; }

};

//===============================

class Man : public Animal

{ public: void function()

{ cout << "Walking ..." << endl; }

};

int main(void)

{

Animal A = Animal();

A.function(); //parent class object

Man m = Man();

m.function(); // child class object

return 0;

}

/\* II. Virtual Function

A virtual function is declared by keyword virtual.

The return type of virtual function may be int, float, void.

A virtual function is a member function in the base class.

We can redefine it in a derived class. It is part of run time

polymorphism. The declaration of the virtual function must be

in the base class by using the keyword virtual.

A virtual function is not static.

The virtual function helps to tell the compiler to perform

dynamic binding or late binding on the function.

If it is necessary to use a single pointer to refer to all

the different classes’ objects. This is because we will have

to create a pointer to the base class that refers to all the

derived objects.

But, when the base class pointer contains the derived class

address, the object always executes the base class function.

For resolving this problem, we use the virtual function.

When we declare a virtual function, the compiler determines

which function to invoke at runtime.

Let’s see the below example for understanding how the program

execution happens without virtual function and with virtual function.

\*/

//without virtual Function

#include <iostream>

using namespace std;

class Add

{

private: int x = 5, y = 20;

public: void display() //overridden function

{

cout << "Value of x is : " << x + y << endl;

}

};

//===================================================================

class Substract : public Add

{

private: int y = 10, z = 30;

public: void display() //overridden function

{ cout << "Value of y is : " << y - z << endl; }

};

//===================================================================

int main()

{

Add\* m; //base class pointer .it can only access the base class members

Substract s; // making object of derived class

m = &s;

m->display(); // Accessing the function by using base class pointer

return 0;

}

#include<iostream>

using namespace std;

class Add

{public: virtual void print()

{ int a = 20, b = 30;

cout << " base class Action is:" << a + b << endl;

}

void show()

{ cout << "show base class" << endl; }

};

//===========================================================

class Sub : public Add

{public: void print() // print () is already virtual function

// in derived class, we could also declared

// as virtual void print () explicitly

{ int x = 20, y = 10;

cout << " derived class Action:" << x - y << endl;

}

void show()

{ cout << "show derived class" << endl; }

};

//======================== main function ======================

int main()

{

Add\* aptr;

Sub s;

aptr = &s;

//==================== Virtual function, binded at runtime (Runtime polymorphism)

aptr->print();

//==================== Non-virtual function, binded at compile time

aptr->show();

return 0;

}

/\* Pure Virtual Function

When the function has no definition, we call such functions as

“Do - nothing function or Pure virtual function”.The declaration

of this function happens in the base class with no definition.

\*/

#include <iostream>

using namespace std;

class Animal

{public:

virtual void show() = 0; //=== Pure virtual function declaration.

};

//================================================

class Man : public Animal

{public:

void show()

{

cout << "Bird is the part of animal family " << endl;

}

};

//================================================

int main()

{

Animal\* aptr; //==== Base class pointer

//Animal a;

Man m; //==== Derived class object creation.

aptr = &m;

aptr->show();

return 0;

}

/\* Difference between run-time and compile-time

Run-time Compile-time

1. Run time is the time 1. Compile time is the time period

period where the executable where the code typed is converted

code is running. into executable code.

2. Errors can be detected only 2. Errors are detected before the execution

after the execution of the of the program.

program. 3. Errors that occur during compile time

3. Errors that occur during are called compile-time errors.

the execution of a program Two types of compile-time errors area)

are called run-time errors. a. semantics error occurs when the

Run time errors aren’t statements aren’t meaningful to the compiler

detected by the compiler. b. Syntax error occurs when the programmer doesn’t

follow the syntax.

A real-time example of polymorphism in C++

A topic can be understood clearly when we compare it to a real time example.

Let’s discuss few real time examples of polymorphism in C++

A person can have various roles and responsibilities at the same time.

A woman plays multiple roles in her life such as a mother, wife, daughter,

daughter in law, sister etc.

A man behaves as an employee in an office, son or husband at home, customer at a mall etc.

A mobile is one device but offers various features such as camera, radio etc.

C++ runtime polymorphism example

\*/

#include <iostream>

using namespace std;

class Animal

{ public: void eat()

{ cout << "Eating"; }

};

class Goat : public Animal

{ public: void eat()

{ cout << "Eating grass"; }

};

int main(void)

{

Goat g = Goat();

g.eat();

return 0;

}

//Run time polymorphism example with data members

#include <iostream>

#include <string>

using namespace std;

class Animal //class declaration.

{ public:

string color = "White";

};

//=====================================

class Rabbit : public Animal // Animal Rabbit class declaration .

{public: string color = "Pink";

};

int main()

{

Animal r = Rabbit();

cout << r.color;

return 0;

}

//Run time polymorphism example using two derived classes

#include <iostream>

using namespace std;

class Polygon

{public: virtual void show()

{ cout << "It is a polygon" << endl; }

};

class Hexagon : public Polygon

{public: void show()

{ cout << "Hexagon is a 6 sided polygon" << endl; }

};

class Pentagon : public Polygon

{public: void show()

{ cout << "Pentagon is a 5 sided polygon" << endl; }

};

//=============================================================

int main()

{

Polygon\* P;

Hexagon h;

Pentagon p;

P = &h;

P->show();

P = &p;

P->show();

return 0;

}