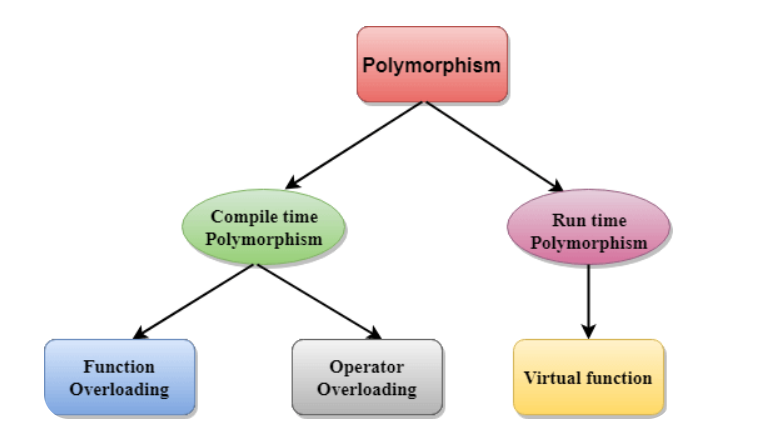
**What is Polymorphism**

The word “poly” means many and “morphs” means forms, So, the word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So, the same person possesses different behaviour in different situations. This is called polymorphism.

****

**Types of polymorphism**

* Compile-time Polymorphism
* Runtime Polymorphism

**Compile-time Polymorphism**

Compile time polymorphism is also known as early binding or static polymorphism. It is performed during compilation time and hence the name compile-time polymorphism.  It is achieved by function overloading and  operator overloading.

**Compile time polymorphism have**

* Function Overloading
* Operator Overloading

**Function Overloading**

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different arguments or return type. Function overloading takes place when you create more than one functions of the same name and these functions serve different purposes.

If we have to perform only one operation and having same name of the functions increases the readability of the program.

**Rules of Function Overloading**

* The functions must have the same name.
* The functions must have different types of parameters.
* The functions must have a different set of parameters.
* The functions must have a different sequence of parameters.

**Syntax**

return\_type funcion\_name (data\_type\_1 variable1, data\_type\_2 variable2, .......) {

//statements

}

**C++ code of function overloading**

#include<iostream>

using namespace std;

class addition{

public:

//we can access public member outside the class but private and

// protected members cannot be accessible outside the class

// Function to add two integers

void add(int a, int b) {

cout<<"Sum of int values : "<<a + b<<endl;

}

// Function to add two doubles

void add(double a, double b) {

cout<<"Sum of double values : "<<a + b<<endl;

}

};

int main() {

addition dd;

// Calls int add(int a, int b)

dd.add(5, 3);

// Calls double add(double a, double b)

dd.add(2.5, 3.7);

return 0;

}

**Operator Overloading**

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.

The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

* Scope resolution operator (::)
* Sizeof operator(sizeof)
* member selector(.)
* member pointer selector(\*)
* ternary operator(?:)

**Syntax**

return\_type operator symbol()

{

//statements

}

**Rules for Operator Overloading**

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

**Types of operator overloading**

* Unary operator overloading
* Binary operator overloading

**Unary Operator Overloading**

The unary operators operate on a single operand and following are the examples of Unary operators −

* [The increment (++) and decrement (--) operators](https://www.tutorialspoint.com/cplusplus/increment_decrement_operators_overloading.htm).
* The unary minus (-) operator.
* The logical not (!) operator.

**Syntax**

class className {

...

public:

returnType operator<symbol>()

{

// custom behaviour for the operator

}

...

};

**C++ code of unary operator overloading with class function**

#include <iostream>

using namespace std;

class Increment {

public:

int x, y;

Increment (int a, int b)

{

x = a;

cout<<"x : "<<x<<endl;

}

//overload + operator

void operator+()

{

x=++x;

cout<<endl<<"After performing unary operator overloading"<<endl;

cout<<"x : "<<x<<endl;

}

};

int main()

{

Increment d1(8, 9);

+d1;

return 0;

}

**C++ code of unary operator overloading with friend function**

#include <iostream>

using namespace std;

class Increment {

public:

int x, y;

Increment (int a, int b)

{

x = a;

cout<<"x : "<<x<<endl;

}

//overload + operator

friend void operator+( Increment &N);

};

void operator+( Increment &N)

{

N.x=++N.x;

cout<<endl<<"After performing unary operator overloading"<<endl;

cout<<"x : "<<N.x<<endl;

}

int main()

{

Increment d1(8, 9);

+d1;

return 0;

}

**Binary Operator Overloading**

The binary operators operate on a two operand and some of Binary operators are +, -, \*, / , % etc.

**Syntax**

class className {

...

public:

returnType operator<symbol>()

{

// custom behaviour for the operator

}

...

};

**C++ code of binary operator overloading with class function**

#include <iostream>

using namespace std;

class add {

private:

int value;

public:

add(int val)

{

value=val;

}

// Overload the + operator

add operator+(add& a) {

int sm = value + a.value;

cout<<"Addition : "<<sm<<endl;

}

};

int main() {

add num1(5);

add num2(10);

add result = num1 + num2;

return 0;

}

**C++ code of binary operator overloading with friend function**

#include <iostream>

using namespace std;

class add {

private:

int value;

public:

add(int val)

{

value=val;

}

// Overload the + operator

friend add operator+(add b, add a);

};

add operator+(add b, add a) {

int sm = b.value + a.value;

cout<<"Addition : "<<sm<<endl;

}

int main() {

add num1(5);

add num2(10);

add result = num1 + num2;

return 0;

}

**Runtime Polymorphism**

Runtime polymorphism is also known as late binding or dynamic polymorphism. It is performed during run time and hence the name run-time polymorphism.  It is achieved by virtual function.

**Runtime polymorphism have**

* **Virtual function**

**Virtual Function/function overriding**

A virtual function is a member function declared in a base class with the virtual keyword, which allows it to be overridden in derived classes. Virtual functions provide a way to achieve dynamic binding or late binding, where the decision about which function to call is made at runtime based on the actual object's type. Function overriding achieve through virtual functions.

**Note : override keyword is optional.**

**Syntax**

class Base {

public:

virtual void someFunction() {

// statements

}

};

class Derived : public Base {

public:

void someFunction() override {

// statements

}

};

int main(){

Base\* basePtr = new Derived;

basePtr->someFunction(); // Calls Derived of someFunction()

}

**C++ code of virtual function/function overriding**

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

virtual void draw() {

cout << "Drawing a shape." <<endl;

}

};

// Derived class 1

class Circle : public Shape {

public:

void draw() override{

cout << "Drawing a circle." <<endl;

}

};

// Derived class 2

class Square : public Shape {

public:

void draw() override{

cout << "Drawing a square." <<endl;

}

};

int main() {

// Create objects of base and derived classes

Shape shape;

Circle circle;

Square square;

// Create pointers to base class objects

Shape\* shapePtr1 = &shape;

Shape\* shapePtr2 = &circle;

Shape\* shapePtr3 = &square;

// Call the draw() function through base class pointers

shapePtr1->draw(); // Output: Drawing a shape.

shapePtr2->draw(); // Output: Drawing a circle.

shapePtr3->draw(); // Output: Drawing a square.

return 0;

}