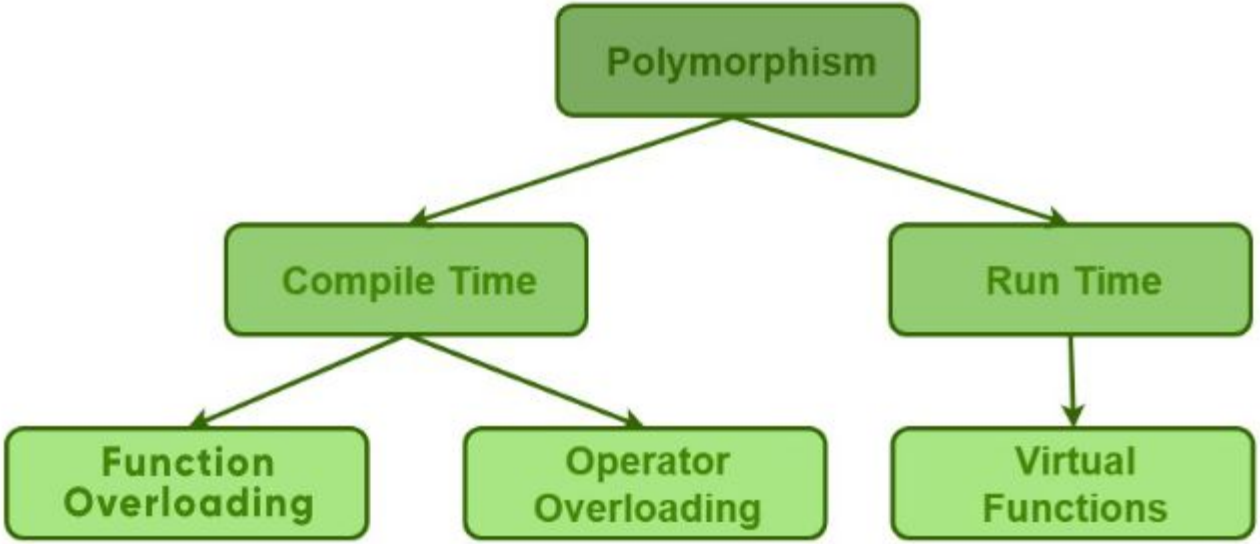
Polymorphism

- The word polymorphism means having many forms.
- It is the ability of a message to be displayed in more than one form.



Function Overloading

- Using this concept of function overloading, We can design a family of a functions with one function name but with different argument lists.
- The function will perform different operations depending on the argument list in the function call.
- For example, add() function can handles different types of data as shown

Function Overloading

```
// Declarations
int add(int a, int b);                          // prototype 1
int add(int a, int b, int c);                   // prototype 2
double add(double x, double y);                 // prototype 3
double add(int p, double q);                    // prototype 4
double add(double p, int q);                    // prototype 5

// Function calls
cout << add(5, 10);                    // uses prototype 1
cout << add(15, 10.0);                 // uses prototype 4
cout << add(12.5, 7.5);                // uses prototype 3
cout << add(5, 10, 15);                // uses prototype 2
cout << add(0.75, 5);                  // uses prototype 5
```

Function Overloading

```
class addition
{
    public:
        void add(int a, int b)
        {   int c;
            c=a+b;
            cout<<"Add_i = "<<c<<endl;
        }
        void add(int a, int b, int c)
        {   c=a+b+c;
            cout<<"Add 3 = "<<c<<endl;
        }
        void add(string s1,string s2)
        {   string s3;
            s3=s1+s2;
            cout<<"String Addition = "<<s3<<endl;
        }
};
```

```
int main()
{
    addition a;
    a.add(10,20);
    a.add(10,20,30);
    a.add("Harish","Gadade");
}
```

Function Overloading

```
class addition
{
    public:
        void add(int a, int b)
        {   int c;
            c=a+b;
            cout<<"Add_i = "<<c<<endl;
        }
        void add(int a, int b, int c)
        {   c=a+b+c;
            cout<<"Add 3 = "<<c<<endl;
        }
        void add(string s1,string s2)
        {   string s3;
            s3=s1+s2;
            cout<<"String Addition = "<<s3<<endl;
        }
};
```

```
int main()
{
    addition a;
    a.add(10,20);
    a.add(10,20,30);
    a.add("Harish","Gadade");
}
```

OUTPUT:

```
30
60
HarishGadade
```

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 that cannot be overloaded are as follows:
    - Scope operator (::)
    - Sizeof
    - member selector(.)
    - member pointer selector(*)
    - ternary operator(?:)

Operator Overloading

Rules for Operator Overloading:

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- When unary operators are overloaded through a member function, it take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Operator Overloading

Syntax of Operator Overloading is

```
return_type class_name  :: operator op(argument_list)
{
    // body of the function.
}
```

Two Types of Operators

- Unary Operators
- Binary Operators

Two Types of Operators Overloading

- Operator Overloading Using Member Functions
- Operator Overloading Using Friend Functions

Unary Operator Overloading Using Member Functions

```
class sample
{
        int a;
        int b;
        int c;
    public:
        void getdata(int a,int y,int z);
        void operator -();
        void display();
};
void sample::getdata(int x, int y, int z)
{
    a=x; b=y; c=z;
}
```

```
void sample::operator -()
{
    a=-a; b=-b; c=-c;
}

void sample::display()
{
    cout<<a<<b<<c;
}

int main()
{   sample s;
    s.getdata(10,-20,30);
    -s;
    s.display();
}
```

Unary Operator Overloading Using friend Functions

```
friend void operator - (sample &S);    // Declaration

void operator - (sample &S)            // Definition
{
     S.a = - S.a;
     S.b = - S.b;
     S.c = - S.c;
}
```

Unary Operator Overloading Using friend Functions

```
class sample
{
        int a;
        int b;
        int c;
    public:
        sample(int x,int y, int z)
        {
            a=x; b=y;  c=z;
        }
        friend void operator -(sample &s)
        {   s.a=-s.a;
            s.b=-s.b;
            s.c=-s.c;
        }
        void display();
};
```

```
void sample::display()
{
    cout<<a<<"  "<<b<<endl;
}
int main()
{
    sample s(10,20,30);
    -s;
    s.display();
    return 0;
}
```

Binary Operator Overloading Using Member Functions

```
class sample
{   private:
        int a;
    public:
        void getdata(int x)
        {
            a=x;
        }

        sample operator +(sample s)
        {
            sample temp;
            temp.a=a+s.a;
            return temp;
        }
```

```
    void display()
    {
          cout<<a<<endl;
    }
};

int main()
{
    sample s1,s2,s3;
    s1.getdata(10);
    s2.getdata(20);
    s3=s1+s2;
    s3.display();
    return(0);
}
```

Binary Operator Overloading Using Friend Functions

```
class sample
{   private:
        int a;
    public:
        void getdata(int x)
        {
            a=x;
        }

        friend sample operator +(sample s1,sample s2)
        {
            sample temp;
            temp.a=s1.a+s2.a;
            return temp;
        }
    void display()
    {
          cout<<a<<endl;;
    }
};

int main()
{
     sample s1,s2,s3;
     s1.getdata(10);
     s2.getdata(20);
     s3=s1+s2;
     s3.display();
     return(0);
}
```