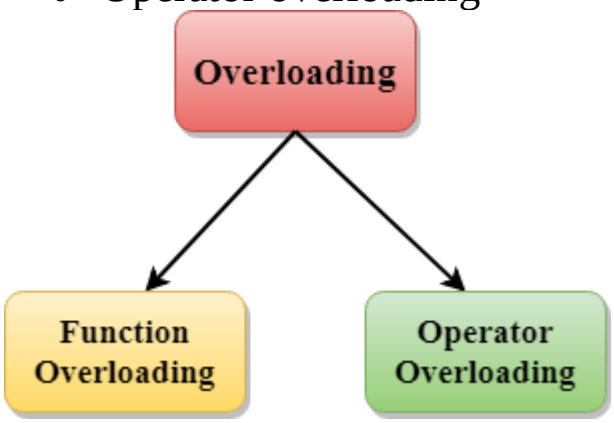
Overloading of functions

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
// program of function overloading when number of arguments vary.
    #include <iostream>
    using namespace std;
```

```
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
Cal C;                                    //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

30

55

Default arguments in C++

In a function, arguments are defined as the values passed when a function is called. Values passed are the source, and the receiving function is the destination.

Definition

A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

Characteristics for defining the default arguments

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Example

- void function(int x, int y, int z = 0)
  Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.

- Void function(int x, int z = 0, int y)
  Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

Code

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
{ // Both z and w are initialised to zero
   return (x + y + z + w); // return sum of all parameter values
}
int main()
{
   cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
   cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
   cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
   return 0;
}
```

Output

25
50
80

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
By using the keyword friend compiler knows the given function is a friend function.
For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
```

```
    friend data_type function_name(argument/s);          // syntax of friend
    function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
```

```
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

```
Length of box: 10
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
      std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
      std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

Output:

Value of x is : 5