

# Function overloading in C++

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function `myfuncn(int a, float b)` is `(int, float)` which is different from the function `myfuncn(float a, int b)` parameter list `(float, int)`. Function overloading is a compile-time polymorphism. Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
sum(int num1, double num2)
```

The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different **type**, **number** or **sequence** of parameters.

## For example:

These two functions have different parameter **type**:

```
sum(int num1, int num2)
sum(double num1, double num2)
```

These two have different **number** of parameters:

```
sum(int num1, int num2)
sum(int num1, int num2, int num3)
```

These two have different **sequence** of parameters:

```
sum(int num1, double num2)
sum(double num1, int num2)
```

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different. For example:

```
int sum(int, int)
double sum(int, int)
```

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

## Function overloading Example

Lets take an example to understand function overloading in C++.

```
#include <iostream>
using namespace std;
class Addition {
public:
    int sum(int num1,int num2) {
        return num1+num2;
    }
    int sum(int num1,int num2, int num3) {
```

```

        return num1+num2+num3;
    }
};
int main(void) {
    Addition obj;
    cout<<obj.sum(20, 15)<<endl;
    cout<<obj.sum(81, 100, 10);
    return 0;
}

```

**Output:**

```

35
191

```

## Function overloading Example 2

As I mentioned in the beginning of this guide that functions having different return types and same parameter list cannot be overloaded. However if the functions have different parameter list then they can have same or different return types to be eligible for overloading. In short the return type of a function does not play any role in function overloading. All that matters is the parameter list of function.

```

#include <iostream>
using namespace std;
class DemoClass {
public:
    int demoFunction(int i) {
        return i;
    }
    double demoFunction(double d) {
        return d;
    }
};
int main(void) {
    DemoClass obj;
    cout<<obj.demoFunction(100)<<endl;
    cout<<obj.demoFunction(5005.516);
    return 0;
}

```

**Output:**

```

100
5006.52

```

## Advantages of Function overloading

The main advantage of function overloading is to improve the **code readability** and allows **code reusability**. In the example 1, we have seen how we were able to have more than one function for the same task (addition) with different parameters, this allowed us to add two integer numbers as well as three integer numbers, if we wanted we could have some more functions with same name and four or five arguments.

Imagine if we didn't have function overloading, we either have the limitation to add only two integers or we had to write different name functions for the same task addition, this would reduce the code readability and reusability.

# Function Overriding in C++

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

## Function Overriding Example

To override a function you must have the same signature in child class. By signature I mean the data type and sequence of parameters. Here we don't have any parameter in the parent function so we didn't use any parameter in the child function.

```
#include <iostream>
using namespace std;
class BaseClass {
public:
    void disp(){
        cout<<"Function of Parent Class";
    }
};
class DerivedClass: public BaseClass{
public:
    void disp() {
        cout<<"Function of Child Class";
    }
};
int main() {
    DerivedClass obj = DerivedClass();
    obj.disp();
    return 0;
}
```

Output:

Function of Child Class

**Note:** In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

## How to call overridden function from the child class

As we have seen above that when we make the call to function (involved in overriding), the child class function (overriding function) gets called. What if you want to call the overridden function by using the object of child class. You can do that by creating the child class object in such a way that the reference of parent class points to it. Lets take an example to understand it.

```
#include <iostream>
using namespace std;
```

```
class BaseClass {
public:
    void disp(){
        cout<<"Function of Parent Class";
    }
};
class DerivedClass: public BaseClass{
public:
    void disp() {
        cout<<"Function of Child Class";
    }
};
int main() {
    /* Reference of base class pointing to
     * the object of child class.
     */
    BaseClass obj = DerivedClass();
    obj.disp();
    return 0;
}
```

Output:

#### Function of Parent Class

If you want to call the Overridden function from overriding function then you can do it like this:

```
parent_class_name::function_name
```

To do this in the above example, we can write following statement in the disp() function of child class:

```
BaseClass::disp();
```

# Difference between Function Overloading and Function overriding in C++

Function overloading and Function overriding both are examples of polymorphism but they are completely different. Before we discuss the difference between them, let's discuss a little bit about them first.

## Function Overloading

Function overloading is a feature that allows us to have same function more than once in a program. Overloaded functions have same name but their signature must be different.

### Example:

Here we have the same function **sum** declared four times with different signatures. Based on the parameters we pass, while calling function **sum**, decides which method is to be called. This happens during compilation, which is why it is also known as compile time polymorphism. If you are wondering why I have suffixed each floating point value with "f" letter in the example below, during function call then refer this: [function overloading float issue](#).

```
#include <iostream>
using namespace std;
// overloaded functions
float sum(int, int);
float sum(float, float);
float sum(int, float);
float sum(float, int);
int main(){
    //This will call the second function
    cout<<sum(15.7f, 12.7f)<<endl;

    //This will call the first function
    cout<<sum(200, 100)<<endl;

    //This will call the third function
    cout<<sum(100, 20.7f)<<endl;

    //This will call the fourth function
    cout<<sum(90.8f, 30)<<endl;

    return 0;
}
float sum(int a, int b){
    return a+b;
}
float sum(float a, float b){
    return a+b;
}
float sum(int a, float b){
    return a+b;
}
float sum(float a, int b){
    return a+b;
}
```

Output:

```
28.4
300
120.7
120.8
```

## Function Overriding

Function overriding is a feature of OOPs Programming that allows us to override a function of parent class in child class.

**Example:**

```
#include<iostream>
using namespace std;
//Parent class or super class or base class
class A{
public:
    void disp() {
        cout<<"Parent Class disp() function"<<endl;
    }
    void xyz() {
        cout<<"xyz() function of parent class";
    }
};
//child class or sub class or derived class
class B : public A{
public:
    /* Overriding disp() function of parent class
     * and giving a different definition to it.
     */
    void disp() {
        cout<<"Child class disp() function"<<endl;
    }
};
int main(){
    //Creating object of child class B
    B obj;
    obj.disp();
    /* If you want to call the overridden function
     * (the same function which is present in parent class)
     * from the child class then assign the reference of
     * parent class to the child class object.
     */
    A obj2 = B();
    obj2.disp();
}
```

Output:

```
Child class disp() function
Parent Class disp() function
```

## Difference between function overloading and function overriding

Now that we understand what is function overloading and overriding in C++ programming, let's see the difference between them:

- 1) Function Overloading happens in the same class when we declare same functions with different arguments in the same class. Function Overriding happens in the child class when child class overrides parent class function.
- 2) In function overloading function signature should be different for all the overloaded functions. In function overriding the signature of both the functions (overriding function and overridden function) should be same.
- 3) Overloading happens at the compile time that's why it is also known as compile time polymorphism while overriding happens at run time which is why it is known as run time polymorphism.
- 4) In function overloading we can have any number of overloaded functions. In function overriding we can have only one overriding function in the child class.

# Virtual functions in C++: Runtime Polymorphism

In this guide, we will see **what are virtual functions and why we use them**. When we declare a function as virtual in a class, all the sub classes that override this function have their function implementation as virtual by default (whether they mark them virtual or not). **Why we declare a function virtual?** To let compiler know that the call to this function needs to be resolved at runtime (also known as **late binding** and dynamic linking) so that the object type is determined and the correct version of the function is called.

Lets take an example to understand what happens when we don't mark a overridden function as virtual.

## Example 1: Overriding a non-virtual function

See the problem here. Even though we have the parent class pointer pointing to the instance (object) of child class, the parent class version of the function is invoked.

You may be thinking why I have created the pointer, I could have simply created the object of child class like this: Dog obj; and assigned the Dog instance to it. Well, in this example I have only one child class but when we a big project having several child classes, creating the object of child class separately is not recommended as it increases the complexity and the code become error prone. More clarity to this after this example.

```
#include<iostream>
using namespace std;
//Parent class or super class or base class
class Animal{
public:
    void animalSound(){
        cout<<"This is a generic Function";
    }
};
//child class or sub class or derived class
class Dog : public Animal{
public:
    void animalSound(){
        cout<<"Woof";
    }
};
int main(){
    Animal *obj;
    obj = new Dog();
    obj->animalSound();
    return 0;
}
```

**Output:**

This is a generic Function



## Example 2: Using Virtual Function

See in this case the output is Woof, which is what we expect. What happens in this case? Since we marked the function animalSound() as virtual, the call to the function is resolved at runtime, compiler determines the type of the object at runtime and calls the appropriate function.

```
#include<iostream>
using namespace std;
//Parent class or super class or base class
class Animal{
public:
    virtual void animalSound(){
        cout<<"This is a generic Function";
    }
};
//child class or sub class or derived class
class Dog : public Animal{
public:
    void animalSound(){
        cout<<"Woof";
    }
};
int main(){
    Animal *obj;
    obj = new Dog();
    obj->animalSound();
    return 0;
}
```

Output:

Woof