Default arguments, Const arguments, Function overloading.

Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.

Following is a simple C++ example to demonstrate the use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

Output:

```
25
50
80
```

When Function overloading is done along with default values. Then we need to make sure it will not be ambiguous.
The compiler will throw error if ambiguous. Following is the modified version of above program.

```
#include<iostream>
```

```
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z=0, float w=0)
{
    return (x + y + z + w);
}
/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

Error:

```
prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl;
                    ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int, int)
 int sum(int x, int y, int z=0, int w=0)
     ^
prog.cpp:10:5: note: candidate:
int sum(int, int, float, float)
 int sum(int x, int y, float z=0, float w=0)
     ^
```

Key Points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.

- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum( int x, int y, int z=0, int w)
```

Constant Argument

A constant argument is the one whose modification cannot take place by the function. Furthermore, in order to make an argument constant to a function, the use of a keyword const can take place like- int sum (const int a, const int b). Moreover, the qualifier const in the function prototype tells the compiler that modification of the argument must not take place by the function.

Understanding Constant Argument in C++

The constant argument turns out to be most useful when the calling of the functions takes place by reference. Furthermore, below is a program that would help in understanding the constant argument and constant variable **C++ (https://www.toppr.com/guides/computer-science/introduction-to-c/)** in the function. Moreover, the program below is extendable and it makes use of pi as a constant argument.

Program to find the area of circle using function with constant argument. It is is important to note that clear difference exists between default arguments and constant arguments in C++.

```
#include

#include

//function prototype

float cir(float,float pi=3.142);

//main function

void main()

{

//clear the screen.

clrscr();

//declare variable as float.
```

```
float r,area;

//Input the radius.

cout<<"Enter the radius"<<endl;

cin>>r;

//calculate area using function.

area=cir(r);

//print area

cout<<"Area of circle is "<<area;

//get character

getch();

}



//function

float cir(float r,float pi)

{

float ar;

ar=pi*r*r;

return(ar);

}
```

Output:

```
Enter the radius.
1
Area of circle is 3.142
```

Its working takes place in three ways which are as follows:

- Entering nter the radius.
- The function calculates the area.
- The printing of the area happens.

C++ Overloading (Function and Operator)

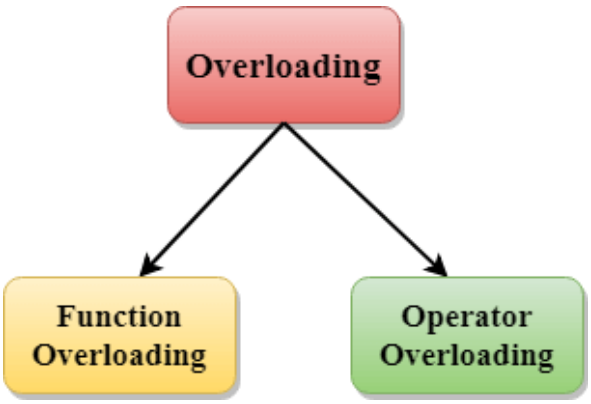
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

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.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2.     public:
3.         return a + b;
4. static int add(int a, int b, int c)
5.         return a + b + c;
6. };
7.     Cal C;                                    //     class object declaration.
8.     cout<<C.add(12, 20, 23);
9. }
```

Output:

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. float mul(float,int);
3.
4. {
5. }
6. {
7. }
8. {
9.     float r2 = mul(0.2,3);
10.     std::cout <<"r2 is : "  <<r2<< std::endl;
11. }
```

Output:

```
r1 is : 42
r2 is : 0.6
```

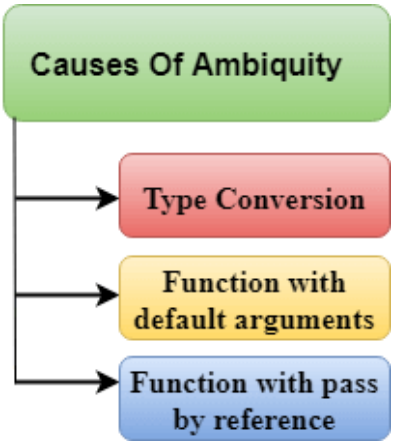
Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



- Type Conversion:

Let's see a simple example.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. void fun(float);
3. {
4. }
5. {
6. }
7. {
8.     fun(1.2);
9. }
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- Function with Default Arguments

Let's see a simple example.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. void fun(int,int);
3. {
```

```
4. }
5. {
6.     std::cout << "Value of b is : " <<b<< std::endl;
7. int main()
8.     fun(12);
9.     return 0;
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

Let's see a simple example.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. void fun(int &);
3. {
4. fun(a); // error, which f()?
5. }
6. {
7. }
8. {
9. }
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

C++ Operators 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. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

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

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. {
2. }
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

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.
- 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.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. {
3.     int num;
```

```
4.       Test(): num(8){}
5.          num = num+2;
6.       void Print() {
7.       }
8. int main()
9.     Test tt;
10.    tt.Print();
11. }
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

(https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#) (https://www.javatpoint.com/cpp-overloading#)

```
1. using namespace std;
2. {
3.     int x;
4.       A(){}
5.     {
6.     }
7.     void display();
8.
9. {
10.    int m = x+a.x;
11.
12. int main()
13.    A a1(5);
14.    a1+a2;
15. }
```

Output:

```
The result of the addition of two objects is : 9
```

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the

function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

(https://www.javatpoint.com/cpp-function-overriding#) (https://www.javatpoint.com/cpp-function-overriding#) (https://www.javatpoint.com/cpp-function-overriding#)

```
1. using namespace std;
2.     public:
3. cout<<"Eating...";
4. };
5. {
6.  void eat()
7.        cout<<"Eating bread...";
8. };
9.    Dog d = Dog();
10.    return 0;
```

Output:

```
Eating bread...
```