Chapter 3 : Function In C++

Call By Reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

For now, let us call the function swap() by passing values by reference as in the following example :

```
#include <iostream>

using namespace std;

void swap(int &x, int &y); // function declaration

int main () {
```

```
// local variable declaration:
int a = 100;
int b = 200;
cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
/* calling a function to swap the values using variable reference.*/
swap(a, b);
cout << " After swap, value of a : " << a << endl;
cout << "After swap, value of b :" << b << endl;
return 0;
```

```
}
```

When the above code is put together in a file, compiled and executed, it produces the following result :

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Return By Reference

Pointers and References in C++ held close relation with one another. The major difference is that the pointers can be operated on like adding values whereas references are just an alias for another variable.

Functions in C++ can return a reference as it's returns a pointer. When function returns a reference it means it returns a implicit pointer.

Return by reference is very different from Call by reference. Functions behaves a very important role when variable or pointers are returned as reference.

For Example :

```
// C++ program to illustrate return by reference
#include <iostream>
using namespace std;
// Function to return as return by reference
int& returnValue(int& x)
{
```

```
// Print the address
cout << "x = " << x<< " The address of x is "<< &x << endl;
    return x;   // Return reference
}
int main() // Driver Code
{
    int a = 20;
    int& b = returnValue(a);
 // Print a and its address
```

```
    cout << "a = " << a<< " The address of a is "<< &a << endl;
   // Print b and its address
    cout << "b = " << b<< " The address of b is "<< &b << endl;
 // We can also change the value of 'a' by using the address returned by returnValue function Since the function returns an alias of x, which is itself an alias of a, we can update the value of a returnValue(a) = 13;
 // The above expression assigns the value to the returned alias as 3.
    cout << "a = " << a<< " The address of a is "<< &a << endl;
    return 0;
 }
```

Output :

x = 20 The address of x is 0x7fff3025711c

a = 20 The address of a is 0x7fff3025711c

b = 20 The address of b is 0x7fff3025711c

x = 20 The address of x is 0x7fff3025711c

a = 13 The address of a is 0x7fff3025711c

Function Overloading and Default Arguments

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

Output :

25

50

80

Inline Function

Inline function is one of the important feature of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function.

This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run.

However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code.

This overhead occurs for small functions because execution time of small function is less than the switching time.

The syntax for defining the function inline is :

```
inline return-type function-name(parameters)
{
    // function code
}
```

Inline function and classes

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

```
#include <iostream>

using namespace std;

class operation
```

```
{
    int a,b,add,sub,mul;
    float div;
public:
    void get();
    void sum();
    void difference();
    void product();
    void division();
};
```

```
inline void operation :: get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
 }
 inline void operation :: sum()
{    add = a+b;
    cout << "Addition of two numbers: " << a+b << "\n";
}
```

```
inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << a-b << "\n";
}
inline void operation :: product()
{
   mul = a*b;
    cout << "Product of two numbers: " << a*b<<"\n";
   }
```

```
inline void operation ::division()
{
    div=a/b;
    cout<<"Division of two numbers:    "<<a/b<<"\n" ;
 }
int main()
{
    cout << "Program using inline function\n";
    operation s;
```

```
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

Output :

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

Default Arguments

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.

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.

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.

Friend Functions

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

For example, a LinkedList class may be allowed to access private members of Node.

• **Following are some important points about friend functions and classes :**

1. Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2. Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.

3. Friendship is not inherited (See this for more details)

4. The concept of friends is not there in Java.

Example :

```
#include <iostream>
class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};
```

```
class B {
private:
    int b;
public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};
```

```
int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

Output :

```
A::a = 0
```

Virtual Functions

A virtual function is a member function which is declared within a base class and is re-defined(Overridden) by a derived class.

When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Following are some important points about Virtual functions :

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be the same in the base as well as derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Example :

```
// CPP program to illustrate
// concept of Virtual Functions
#include <iostream>
using namespace std;
class base {
public:
  virtual void print()
  {
        cout << "print base class" << endl;
```

```
    }
   void show()
    {
        cout << "show base class" << endl;
    }
};
class derived : public base {
public:
 void print()
    {
```

```
        cout << "print derived class" << endl;
    }
 void show() {
 cout << "show derived class" << endl;
    }
};
int main() {
    base* bptr;
    derived d;
    bptr = &d;
```

```
    bptr->print();  // virtual function, binded at runtime
    bptr->show();  // Non-virtual function, binded at compile time
}
```

Output :

```
print derived class
show base class
```

Difference Between Friend & Virtual Functions

Friend Function	Virtual Function
It is non-member functions that usually have private access to class representation.	It is a base class function that can be overridden by a derived class.
It is used to access private and protected classes.	It is used to ensure that the correct function is called for an object no matter what expression is used to make a function class.
It is declared outside the class scope. It is declared using the ‘friend’ keyword.	It is declared within the base class and is usually redefined by a derived class. It is declared using a ‘virtual‘ keyword.

Friend Function	Virtual Function
It is generally used to give non-member function access to hidden members of a class.	It is generally required to tell the compiler to execute dynamic linkage of late binding on function.
They support sharing information of class that was previously hidden, provides method of escaping data hiding restrictions of C++, can access members without inheriting class, etc.	They support object-oriented programming, ensures that function is overridden, can be friend of other function, etc.
It can access private members of the class even while not being a member of that class.	It is used so that polymorphism can work.