1.6 FUNCTIONS IN C++

Definition

A **function** in C++ is a named block of reusable code that performs a specific task. It can be executed whenever needed in a program. Functions help achieve modularity, reduce repetition, and make programs easier to maintain.

Need for Functions

- Large programs become easier to understand when divided into smaller modules.
- Code can be reused multiple times without rewriting the same logic.
- Simplifies testing, debugging, and maintenance.
- Enhances teamwork by allowing multiple programmers to work on separate functions.

1.6.1 Function Declaration (Prototype)

A function declaration tells the compiler about:

- The function name
- Return type
- Parameter list

Syntax:

```
return_type function_name(parameter_list);
```

It prevents errors by ensuring the function is called correctly.

Function Definition

This contains the actual body of the function where the logic is written.

Syntax:

```
return_type function_name(parameter_list)
{
   // body of function
}
```

Function Call

A function is executed using its name followed by parentheses.

Syntax:

```
function_name(arguments);
```

Example Program (Function Declaration, Definition, Call)

```
#include <iostream.h>
class Calculator
{
    public:
   int add(int a, int b)                                    // Member function
  {
        return a + b;
  }
};

void main()
{
     Calculator c;

     int result = c.add(10, 20);           // Function call through object
     cout << "Sum = " << result;

}
```

Output:

```
30
```

Advantages

- Makes the program modular and easy to understand.
- Avoids repetition of code.
- Easy debugging and testing.
- Enhances reusability because the same function can be used anywhere.
- Facilitates teamwork by dividing the program into tasks.

Disadvantages

- Too many small functions may reduce program efficiency.
- Passing large data by value increases memory usage.
- Program flow becomes harder to trace with many function calls.
- Overuse of functions may fragment the logic.

Example

```
#include <iostream.h>
class Multiply
{
   public:
       int multiply(int a, int b)                                  // Member function
      {
             return a * b;
       }
};

void main()
{
    Multiply m;                                  // Object name starts with 'M' → m
```

```
        cout << "Product = " << m.multiply(4, 5);
    }
```

1.6.3 Function Overloading

Function overloading allows **multiple functions with the same name** but with **different parameter lists** (type or number of parameters).

Syntax

```
return_type function_name(type1);
return_type function_name(type1, type2);
```

Example

```
#include <iostream.h>
class Addition
{
   public:
      int add(int a)             // Function with one argument
      {
         return a + 0;
      }

      int add(int a, int b)      // Function with two arguments
      {
         return a + b;
      }
};

void main( )
{
   Addition A1;                  // Object creation
   cout << "Sum (two numbers) = " << A1.add(4, 5) << endl;
```

```
        cout << "Sum (one number) = " << A1.add(6);
    }
```

1.6.4 Inline Functions

An inline function in C++ is a function whose **function call is replaced by the actual function code at compile time**, thereby reducing function-call overhead and improving execution speed.

The inline keyword suggests the compiler to **expand the function code at the point of call**, which is especially useful for **small and frequently used functions**.

Inlining is **only a request**, not a command; the compiler may choose or ignore it.

The compiler may ignore the inline request if the function:

- Contains **loops**
- Uses **recursion**
- Contains **static variables**
- Uses **switch or goto statements**
- Is a **non-void function without a return statement**

Syntax

```
inline return_type function_name(parameter_list)
{
   // body
}
```

Example

```
#include <iostream.h>

class Addition

{

public:
    inline int add(int a, int b)          // Inline member function
    {
        return a + b;
    }
};
void main()
{
    Addition A1;                                   // Object name starts with
'A' → A1
    cout << "Addition = " << A1.add(10,20);
}
```

Advantages:

- Faster execution (no function call overhead).
- Useful for small, frequently called functions.

Disadvantages:

- Increases program size if used excessively.
- Not suitable for large functions or those with loops and recursion.

1.6.5 Default Arguments

Default arguments allow a function to assign **default values** to parameters if no value is provided during the function call.

Syntax

```
return_type function_name(int x, int y = 10);
```

Example

```
#include <iostream>
```

```
class Addition
{
public:
    int add(int a, int b = 5)                           // Default argument
    {
        return a + b;
    }
};
void main( )
{
    Addition a;                                   // Object name starts with 'A' → a
    cout << a.add(10) << endl;                          // Uses default b = 5
    cout << a.add(10, 20);                               // Overrides default
}
```

Advantages

- Flexibility in calling functions.
- Reduces the number of overloaded functions needed.

Disadvantages

- Can cause confusion when too many defaults are used.
- Order of parameters becomes important.