**Functions**

### ****What is a Function in C++?****

In **C++**, a **function** is a **block of code (Set of instructions)** with the unique name that performs a specific task. Functions help organize code, avoid repetition, and make programs easier to read and maintain.

### ****Basic Structure of a Function [Function Declaration and definition or Function prototype]****

return\_type function\_name(Formal parameters)

{

// body of the function

}

**Function Calling**

### Variable\_name = function\_name (actual parameters) ;

**Types of Functions**

1. **Built-in functions** – Provided by C++ (e.g., pow () sqrt(), strlen()).
2. **User-defined functions** – Created by the programmer (like add(), calculate()).

**Why Use Functions?**

* To reuse code (write once, use many times).
* To make code modular and readable.
* To break complex problems into smaller parts.

### ****Example program for function concept****

// Programs using function

#include <iostream>

using namespace std;

// Function declaration and definition

int add(int a, int b)

{

return a + b;

}

void display( )

{

cout << “Welcome to Function Concept in Programming “;

}

void multiply(int x, int y) // Formal parameters

{

int multi;

multi = x \* y;

cout << “Multiplication = “ << multi << endl;

}

int main()

{

int a, b;

cout << “Enter 2 numbers :: “ ;

display();

int result = add(a, b); // Function call

cout << "Sum is: " << result;

multiply ( a, b ); // Actual parameters

return 0;

}

**Example for function**

// Example C++ Program for Function concept

#include <iostream>  
using namespace std;

void welcome()  // Function declaration and definition OR Function //prototype - Its a function with  No arguments and no return //datatype  
{  
    cout << "Welcome to Function concept \n";  
}

void displayName (char\* name) // Function with 1 input arg and no //return value  // Formal arguments  
{  
    cout << "My name is " << name << "\n";  
}

int checkAge (char\* my\_name, int age ) // Function with 2 input //arguments and returns a value  
{  
    cout << "My age = " << age << "\n";  
    welcome();  
    if (age >= 18 )  
        return 1;  
    else  
        return 0;  
}

int addAge (int age)  
{  
   int incAge;  
   incAge = age + 10;  
   return incAge;  
}

int main() {  
    char my\_name[50] = {"Harsha" };  
    int age = 23;  
    int result;  
     welcome();  // Function call  
     displayName(my\_name); // Actual arguments  
  
     result = checkAge(my\_name, age);  
     if (result == 1)  
        cout << my\_name << "  is major \n";  
     else  
        cout << my\_name <<"  is minor \n";  
  
    cout << "Increased Age = " << addAge(age) << "\n";  
    welcome();  
    return 0;  
}

**What is an Inline Function in C++?**

An **inline function** in C++ is a function which proceed with **‘inline’** keyword where the **compiler replaces the function call with the actual code of the function**, to avoid the overhead of a function call.

**Syntax**

**inline return\_type function\_name(parameters) {**

**// function body**

**}**

**Example for ‘inline’ function**

// Example for inline function

#include <iostream>

using namespace std;

inline int square(int x) {

return x \* x;

}

int main() {

cout << "Square of 5 is: " << square(5) << endl;

return 0;

}

**Why Use Inline Functions?**

* Speeds up execution by **avoiding function calls**.
* Good for **small, frequently used functions**.

**When Not to Use Inline Functions**

* In large functions (increases code size).
* In recursive functions.
* When using virtual functions (in polymorphism).

**Note:**

* It's only a **request** to the compiler — **not a command**. The compiler may ignore inline if:
  + The function is too complex.
  + It contains loops, recursion, or static variables.

### ****Inline Function vs Regular Function****

| **Feature** | **Inline Function** | **Regular Function** |
| --- | --- | --- |
| **Definition** | Function code is expanded at the point of call | Function call is made and control jumps to function body |
| **Speed** | Faster (no function call overhead) | Slightly slower (due to call and return mechanism) |
| **Memory Usage** | May increase memory (code duplication) | More memory efficient (code resides in one place) |
| **Usage** | Best for small, frequently used functions | Suitable for large or complex functions |
| **Syntax** | Uses **inline** keyword | No special keyword needed |
| **Compiler Control** | Only a request to compiler; may be ignored | Always handled by compiler with function calls |
| **Debugging** | Harder to debug (no actual function call stack) | Easier to debug (step into function call) |
| **Recursion** | Not allowed or discouraged | Fully supported |

### ****What is Recursion in C++?****

**Recursion** is a programming technique where a **function calls itself** to solve a problem.

It is often used to solve problems that can be **broken down into smaller, similar sub-problems**.

NOTE: In recursion the recursive function is called again and again until the specific condition is reached.

NOTE: When recursion is used “STACK [Lost in First Out]” register is used to hold all the

return values of the recursive function call.

**Basic Structure of a Recursive Function**

*return\_type function\_name(parameters) {*

*if (base\_condition) {*

*// stop calling itself*

*return some\_value;*

*} else {*

*// recursive call*

*return function\_name(modified\_parameters);*

*}*

*}*

**Example: Factorial using Recursion**

// Program to find factorial using recursion

#include <iostream>

using namespace std;

int factorial(int n) {

if (n == 0) // base condition

return 1;

else

return n \* factorial(n - 1); // recursive call

}

int main() {

cout << "Factorial of 5 is: " << factorial(5);

return 0;

}

**Key Terms**

* **Base Case:** Stops the recursion (e.g., if(n == 0)).
* **Recursive Case:** The function calls itself with modified input.

**Advantages of Recursion**

* Simplifies code for problems like:
  + Factorial
  + Fibonacci series
  + Tree traversal
  + Tower of Hanoi
  + Data structure related problems (Arrays, Stacks, Queues, Linked List, Trees etc)

**Disadvantages**

* Can lead to **stack overflow** if not controlled.
* May be **slower** and **use more memory** than loops in some cases.

**WHAT IS FUNCTION OVERLOADING IN C++?**

**Function Overloading** in C++ means having **multiple functions with the same name** but with the different signatures. That means function overloading is the concept of having more than one functions with the same name but the number of parameters, type of parameters or the order of parameters are different.

When all the signatures are same, the compiler will throw an error. The compiler don’t bother about the return data type of the parameter.

It allows a programmer to use the **same function name** to perform **different tasks**, based on the **arguments passed** to it.

NOTE: **Function overloading** means **defining multiple functions with the same name but different parameters** (type or number). The correct function is selected by the compiler based on the arguments passed.

NOTE: Function Overloading is a type of polymorphism. It is static polymorphism.

### ****Polymorphism**** in C++ (and other object-oriented languages) means "****many forms****."

It allows the same function or object to behave differently based on the context.

### ****Types of Polymorphism in C++:****

#### 1. **Compile-Time Polymorphism (Static Binding):**

Achieved using:

* **Function Overloading**
* **Operator Overloading**

#### **Run-Time Polymorphism (Dynamic Binding):**

Achieved using:

* **Virtual Functions**

### Why Use Polymorphism?

* Flexibility and extensibility in code
* Enables **generic programming**
* Reduces code duplication
* Supports **runtime decision-making**

**KEY POINTS:**

* Functions must differ in **number or type or order of parameters**.
* It helps in improving **code readability and reusability**.
* **Return type alone** cannot be used to distinguish overloaded functions. [Return data type is not important in function overloading]

// Example for Function Overloading concept

#include <iostream>

using namespace std;

// Function to add two integers

int add(int a, int b) {

return a + b;

}

// Function to add two floats

float add(float a, float b) {

return a + b;

}

// Function to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

// void display(string name, int age)

{

cout<< “Age of “ << name << “ is ” << age << “\n”;

}

// void display(int age, string name)

{

cout<< “Age of “ << name << “ is ” << age << “\n”;

}

int main() {

cout << "add(3, 4) = " << add(3, 4) << endl; // Calls int add(int, int)

cout << "add(2.5f, 4.5f) = " << add(2.5f, 4.5f) << endl; // Calls float add(float, float)

cout << "add(1, 2, 3) = " << add(1, 2, 3) << endl; // Calls int add(int, int, int)

cout << display(“Milton”, 24);l

cout << display(20, “Franco”);

return 0;

}

**Example for Virtual Function**

### #include <iostream>

### using namespace std;

### class Animal {

### public:

### virtual void sound() {

### cout << "Animal makes a sound" << endl;

### }

### };

### class Dog : public Animal {

### public:

### void sound() override {

### cout << "Dog barks" << endl;

### }

### };

### int main() {

### Animal\* a;

### Dog d;

### a = &d;

### a->sound(); // Calls Dog's version (dynamic dispatch)

### return 0;

### }

**Function with default arguments**

In C++, **functions with default arguments** allow you to **assign default values to parameters**, so when the function is called with fewer arguments, the defaults are automatically used.

### Syntax:

return\_type function\_name(parameter1 = value1, parameter2 = value2, ...);

**Important Note:**

**Default values must be assigned from right to left** (i.e., trailing parameters only).

**Example:**

#include <iostream>

using namespace std;

void greet(int empCode, string name = "Guest", int age = 18) {

cout << "Hello, " << name << "! You are " << age << " years old." << endl;

}

int main() {

greet(101); // Uses both default values

greet(1002,"Alice"); // Uses default age

greet(345,"Bob", 25); // Uses no defaults

greet( ); // No actual arguments, Error

return 0;

}

### ****Static Variable with Function Call – Example in C++****

A **static variable inside a function** retains its **value between function calls**. It is initialized **only once** and **does not get destroyed** when the function exits. Static variable is initialised once, from the next time onwards the current value is taken for execution.

// Example for **Static Variable with Function Call**

#include <iostream>

using namespace std;

void countCalls() {

static int count = 0; // static variable retains value

count++;

cout << "Function called " << count << " times." << endl;

}

void nonStaticCalls() {

int count = 0; // Non-static variable with value

count++;

cout << "Function called " << count << " times." << endl;

}

int main() {

countCalls(); // 1st call

countCalls(); // 2nd call

countCalls(); // 3rd call

nonStaticCalls();

nonStaticCalls();

nonStaticCalls();

return 0;

}

**Output:**

Function called 1 times.

Function called 2 times.

Function called 3 times.

Function called 1 times.

Function called 1 times.

Function called 1 times.

**Key Points:**

* static int count = 0; is **initialized only once**, even though the function is called multiple times.
* Each time the function is called, the **previous value of count is remembered/retained**.
* Static variables have **function scope** but **lifetime throughout the program execution**.