
# *Object Oriented Programming using C++*


# *Recursion*


# *Recursion in Functions*

A function is said to be recursive if it calls itself within its own body. Recursion is a powerful technique that allows a function to solve problems by breaking them down into smaller, more manageable instances of the same problem.

1. Definition of Recursion:

- Recursive Function: A function that makes one or more calls to itself in its definition.

- Circular Definition: Recursion involves defining something in terms of itself, which can be thought of as a circular definition.

2. Example of Recursion:

```
int fact(int n) {
    int ans;
    if (n == 1) // Base case
        return 1;
    ans = fact(n - 1) * n; // Recursive call
    return ans;
}
```


3. Function Execution and Stack:

- When a recursive function is called, a new set of local variables and parameters are allocated on the call stack for each call.

- Each call to the function operates with its own set of variables, and when the function completes, those local variables are removed from the stack.

- Execution resumes at the point of the function call in the previous call, using the results of the recursive call.

4. Memory and Execution:

- Stack Allocation: Each recursive call allocates new space on the stack. This stack space holds local variables and parameters specific to that call.

- Stack Unwinding: When a recursive call completes, its stack space is freed, and execution continues from the point where it was called.

# *Recursive Function Characteristics*

- Efficiency: Recursive functions can be elegant and reduce code complexity, but they may also lead to performance issues if not properly managed (e.g., excessive stack usage).

- Termination: It's crucial to have a base case to avoid infinite recursion, which can lead to a stack overflow.


# *Example of Factorial Calculation*

```
#include <stdio.h>
    int fact(int n) {
        if (n == 1) // Base case
            return 1;
        else
            return fact(n - 1) * n; // Recursive case
    }
    int main(void) {
        int number = 5;
        printf("Factorial of %d is %d\n", number, fact(number));
        return 0;
    }
```

Output:

Factorial of 5 is 120

# *Inline functions*

Inline functions in C++ are functions defined with the `inline` keyword, suggesting to the compiler that it should attempt to insert the function's code directly into the places where the function is called. This approach can help to reduce the overhead associated with function calls, such as jumping to the function and saving registers.

- Purpose: To minimize the overhead of function calls by integrating the function's code directly at each call site.

- Syntax: Prefix the function definition with the `inline` keyword.

- Advantage: Offers type safety and allows for better debugging compared to macros, which only provide code replacement without type checking.

- Usage: Typically used for small, frequently called functions to improve performance.


Example:

```
class MyClass {
public:
    inline void display() const {
        std::cout << "Display function" << std::endl;
    }
};
```

In this example, the `display` function is defined as `inline`, which suggests to the compiler to insert the function's code at each call site, potentially improving execution speed.


# *When to use inline functions ?*

Guidelines for Using Inline Functions

- General Advice: Inline functions should generally be used sparingly. Overuse can lead to larger executable sizes and potentially more complex debugging.

- When to Use: Consider using inline functions if a fully developed and tested program runs too slowly, and if the function's execution time is significant compared to the overhead of a function call.

- Ideal Use Cases: Inline functions are particularly beneficial when the function is simple, such as those with a single return statement or a few lines of code.

- Performance Considerations: Inline functions should be implemented when the time saved by avoiding the function call overhead outweighs the cost of increasing the function's code size in the compiled output.


Example:

```
// Inline function for simple return
inline int square(int x) {
    return x * x;
}
```

In this case, `square` is a good candidate for inlining because it is a small, simple function. However, inline functions should be used judiciously, ensuring they provide a real performance benefit without unnecessarily inflating the size of the compiled program.

# Default function Arguments

- Function Declaration: In C++, you can call a function without specifying all its arguments. Default values for parameters are provided in the function declaration.

- Compiler Behavior: The compiler uses the function prototype to determine the number of arguments and applies default values if they are not provided during the function call.

- Syntax: Default values are set in the function declaration, similar to variable initialization.

Example:

```
// Function declaration with default values

float amount(float principal, int time, float rate = 9.5);

// Function definition

float amount(float principal, int time, float rate) {

   return principal * time * rate;

}

// Function calls

float a1 = amount(1000.0, 5);       // Uses default rate of 9.5

float a2 = amount(1000.0, 5, 10.0);  // Uses specified rate of 10.0
```