Functions in CPP

Functions are a block of named code that can be called to perform a specific task.
Functions give us a variety different use cases that help in reducing the line of code and improving the readability of the code.
Following points to get the concept of functions clear:

1. Types of Functions:
  - With Return Type:
    - Functions that return a value to the calling function.
    - Declared using a return type (e.g., `int`, `float`).

  - **Without Return Type (void):
    - Functions that do not return any value.
    - Declared using `void` as the return type.

2. Parameterization:
  - Parameterized Functions:
    - Functions that accept parameters.
    - Parameters provide input to the function.
    - Example: `int add(int num1, int num2) { return num1+ num2; }`

  - Non-Parameterized Functions:
    - Functions that do not take any parameters.
    - Example: `void greet() { printf("Hello!"); }`

3. Call by Value & Call by Reference:
  - Call by Value:
    - Parameters passed by value.
    - Function works with a copy of the actual parameters.
    - Changes inside the function do not affect the original values , as it works with copy of the parameters.

  - Call by Reference:
    - Parameters passed by reference (using pointers).
    - Function takes input of address as parameter and pointer as utilised to deference them inside the function.
    - Function works with the actual memory locations of the parameters.
    - Changes inside the function affect the original values.

Advantages of functions:

- Functions provide modularity and reusability in C programming.

- Use return types to indicate the type of value returned (if any).
- Parameters allow passing information to functions for processing.
- Call by Value makes a copy of the values, while Call by Reference works with actual memory locations.

Function Overloading in C++

Definition:
Function overloading is a feature in C++ that allows multiple functions with the same name but different parameters or parameter types to coexist within the same scope.

Key Points:

1. Same Function Name:
   - Multiple functions with the same name are defined.
   - Differentiated based on the type and/or number of parameters.

2. Parameter Variation:
   - Overloaded functions have distinct parameter lists.
   - Parameters may differ in type, number, or both.

3. Return Type Irrelevant:
   - Overloaded functions may have the same or different return types.
   - The compiler distinguishes functions primarily based on parameters.

4. Example:

```
// Function Overloading Example
#include <iostream>

// Overloaded functions
int add(int n1, int n2) {
    return n1 + n2;
}

double add(double n1, double n2) {
    return n1+ n2;
}

// Usage
int main() {
    std::cout << "Sum (int): " << add(5, 10) << std::endl;
    std::cout << "Sum (double): " << add(5.5, 10.5) << std::endl;
    return 0;
```

```
}
```

5. Compile-Time Resolution:
   - The compiler determines the appropriate function to call during compile-time based on the provided arguments.

6. Enhances Readability:
   - Improves code readability by using intuitive function names.
   - Eliminates the need for cryptic names to differentiate functions.

7. Restrictions:
   - Overloaded functions must have unique parameter lists; return types alone are insufficient for differentiation.

8. Not Supported in C:
   - Function overloading is a feature specific to C++ and is not supported in the C programming language.

Function overloading simplifies code organization and enhances code readability by allowing multiple functions with similar functionality to share a common name while accommodating different data types or parameter counts.

Inline Functions:

- Definition:
  -Purpose: Suggests the compiler to insert the complete function body at the call site.
  - Keyword: Declared using the `inline` keyword before the function declaration.

- Usage Guidelines:
  - Effective for small, frequently called functions.
  - Avoid for large functions to prevent code bloat.

- Example:

```
// Inline Function Example
inline int add(int a, int b) {
    return a + b;
}
```

- Benefits:
  - Improves performance for small functions.
  - Reduces function call overhead.

- Considerations:
  - Compiler decision on whether to honor the `inline` request.
  - Use with care to balance size and performance.

Virtual Functions:

- Definition:
  - Purpose: Enables polymorphism by allowing a function to be overridden in derived classes.
  - Keyword: Declared using the `virtual` keyword in the base class.

- Usage Guidelines:
  - Base class provides a common interface.
  - Derived classes override virtual functions.

- Key Concepts:
  - Late binding: Function resolution occurs at runtime.
  - Dynamic polymorphism: Objects of different types treated through a common interface.

- Additional Keywords:
  - C++11 onwards, explicitly indicates overriding a virtual function.
  - pure virtual function: A virtual function with no implementation, making the class abstract.

- Destructor Consideration:
  - Often includes a virtual destructor for proper cleanup.

These features, inline functions and virtual functions, provide optimization and flexibility in C++ programming, each serving distinct purposes in different contexts.