C++ Functions Guide

Complete Reference: Types, Syntax & Best Practices

1. Introduction to Functions

A function is a reusable block of code that performs a specific task. Functions help in organizing code, improving readability, and promoting code reuse. In C++, functions must be declared before use and can be defined separately.

2. Basic Function Syntax

```
return_type function_name(parameter_list) {
    // Function body
    // Statements
    return value; // if return_type is not void
}
```

3. Types of Functions

3.1 Standard Functions

Regular functions with return values and parameters.

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3);
    cout << result; // Output: 8
    return 0;
}
```

3.2 Void Functions

Functions that don't return any value.

```
void greet(string name) {
    cout << "Hello, " << name << "!" << endl;
}

int main() {
    greet("Alice"); // Output: Hello, Alice!
    return 0;
}
```

3.3 Inline Functions

Functions expanded at compile time to reduce function call overhead. Best for small, frequently-called functions.

```
inline int square(int x) {
    return x * x;
}

int main() {
    cout << square(5); // Output: 25
    return 0;
}
```

> 💡 **Tip:**
>
> Use inline functions only for small functions (1-3 lines). The compiler may ignore the inline request for complex functions.

3.4 Recursive Functions

Functions that call themselves to solve problems by breaking them into smaller subproblems.

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

int main() {
    cout << factorial(5); // Output: 120
    return 0;
}
```

⚠️ **Warning:**

Always include a base case in recursive functions to prevent infinite recursion and stack overflow.

3.5 Function Overloading

Multiple functions with the same name but different parameters (number or type).

```
int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}
```

```
int main() {
    cout << add(2, 3) << endl;          // Output: 5
    cout << add(2.5, 3.7) << endl;      // Output: 6.2
    cout << add(1, 2, 3) << endl;       // Output: 6
    return 0;
}
```

3.6 Default Arguments

Functions can have default values for parameters if not provided by the caller.

```
void display(string msg = "Hello", int times = 1) {
    for (int i = 0; i < times; i++) {
        cout << msg << endl;
    }
}

int main() {
    display();              // Uses defaults: Hello (once)
    display("Hi");          // Uses default times: Hi (once)
    display("Hey", 3);      // Custom values: Hey (3 times)
    return 0;
}
```

💡 **Tip:**

Default arguments must be specified from right to left in the parameter list.

3.7 Pass by Value

A copy of the argument is passed to the function. Changes don't affect the original variable.

```
void modify(int x) {
    x = 100;
}
```

```
int main() {
    int num = 50;
    modify(num);
    cout << num; // Output: 50 (unchanged)
    return 0;
}
```

3.8 Pass by Reference

The actual variable is passed to the function. Changes affect the original variable.

```
void modify(int &x) {
    x = 100;
}

int main() {
    int num = 50;
    modify(num);
    cout << num; // Output: 100 (changed)
    return 0;
}
```

3.9 Pass by Pointer

The address of a variable is passed to the function.

```
void modify(int *x) {
    *x = 100;
}

int main() {
    int num = 50;
    modify(&num);
    cout << num; // Output: 100 (changed)
    return 0;
}
```

3.10 Const Parameters

Parameters that cannot be modified inside the function.

```
void display(const int &value) {
    cout << value << endl;
    // value = 10; // ERROR: cannot modify const parameter
}

int main() {
    int num = 42;
    display(num);
    return 0;
}
```

3.11 Lambda Functions (C++11)

Anonymous functions defined inline for quick operations.

```
int main() {
    auto add = [](int a, int b) -> int {
        return a + b;
    };

    cout << add(5, 3); // Output: 8

    // Lambda with capture
    int multiplier = 3;
    auto multiply = [multiplier](int x) {
        return x * multiplier;
    };

    cout << multiply(4); // Output: 12
    return 0;
}
```

3.12 Template Functions

Generic functions that work with any data type.

```
template
T getMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << getMax(10, 20) << endl;       // Output: 20
    cout << getMax(15.5, 12.3) << endl;   // Output: 15.5
    cout << getMax('a', 'z') << endl;     // Output: z
    return 0;
}
```

4. Function Prototypes

Function declarations that inform the compiler about function existence before actual definition.

```
// Function prototype
int multiply(int, int);

int main() {
    int result = multiply(4, 5);
    cout << result; // Output: 20
    return 0;
}

// Function definition
int multiply(int a, int b) {
    return a * b;
}
```

5. Return Types

5.1 Returning by Value

```
int getValue() {
    return 42;
}
```

5.2 Returning by Reference

```
int& getElement(int arr[], int index) {
    return arr[index];
}

int main() {
    int data[5] = {1, 2, 3, 4, 5};
    getElement(data, 2) = 99;
    cout << data[2]; // Output: 99
    return 0;
}
```

⚠️ **Warning:**

Never return a reference to a local variable as it will be destroyed after function exits.

5.3 Returning Pointers

```
int* createArray(int size) {
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
    }
    return arr;
}

int main() {
    int* myArray = createArray(5);
    // Use the array
    delete[] myArray; // Don't forget to free memory!
    return 0;
}
```

6. Best Practices & Tips

💡 Function Naming:

Use descriptive verb-noun combinations: calculateTotal(), getUserInput(), validateEmail().

💡 Single Responsibility:

Each function should do one thing and do it well. If a function is too long, consider breaking it down.

💡 Pass by Reference for Large Objects:

Use const references for large objects to avoid copying: void process(const vector<int> &data).

💡 Use const Whenever Possible:

Mark functions as const if they don't modify member variables (in classes).

⚠️ Avoid Global Variables:

Pass data through function parameters instead of using global variables.

💡 Function Length:

Keep functions short (ideally under 50 lines). Long functions are hard to understand and maintain.

7. Common Pitfalls to Avoid

- Forgetting to return a value in non-void functions
- Returning references or pointers to local variables
- Not handling edge cases in recursive functions
- Excessive function overloading leading to confusion
- Not freeing dynamically allocated memory returned by functions
- Modifying const parameters or const objects
- Missing function prototypes when definitions come after usage

8. Advanced Concepts

8.1 Function Pointers

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    int (*operation)(int, int);

    operation = add;
    cout << operation(5, 3) << endl; // Output: 8
    operation = subtract;
    cout << operation(5, 3) << endl; // Output: 2

    return 0;
}
```

8.2 Constexpr Functions (C++11)

Functions evaluated at compile time for constant expressions.

```
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

int main() {
    const int result = factorial(5); // Computed at compile time
    int arr[factorial(4)]; // Can be used for array size
    return 0;
}
```

8.3 Variadic Functions

Functions that accept variable number of arguments.

```
#include

int sum(int count, ...) {
    va_list args;
    va_start(args, count);

    int total = 0;
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int);
    }

    va_end(args);
    return total;
}

int main() {
    cout << sum(3, 10, 20, 30) << endl; // Output: 60
    cout << sum(5, 1, 2, 3, 4, 5) << endl; // Output: 15
    return 0;
}
```

9. Quick Reference Table

Type	Use Case	Example
Inline	Small, frequently called functions	`inline int square(int x)`
Recursive	Problems with repetitive structure	`int factorial(int n)`
Template	Generic, type-independent code	`template<typename T>`
Lambda	Quick, anonymous operations	`[](int x) { return x*2; }`
Overloaded	Same operation, different types	`add(int), add(double)`

10. Conclusion

Functions are fundamental building blocks in C++ programming. Mastering different types of functions and their appropriate usage will significantly improve your code quality, maintainability, and efficiency. Practice implementing various function types and always strive for clean, readable code.