Functions in C++

1. Introduction to Functions

In C++, a function is a block of code that performs a specific task. Functions help in organizing code into logical units, promoting reusability, modularity, and readability.

Every C++ program has at least one function: main(). You can define additional functions to break down complex problems.

2. Function Declaration, Definition, and Call

Function Declaration (Prototype)

Tells the compiler about the function's name, return type, and parameters.

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

Example:

```
int add(int a, int b);
```

Function Definition

Contains the actual implementation of the function.

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

2. Function Declaration, Definition, and Call

Function Call

Invokes the function to execute its code.

```
int result = add(3, 5); // result is 8
```

Note: A function must be declared before it is called unless it is defined before the call.

3. Function Parameters and Arguments

- Parameters are variables listed in the function declaration.
- Arguments are actual values passed to the function when it is called.

C++ supports three main ways to pass arguments:

1. Pass by Value – A copy of the argument is passed. Changes inside the function do not affect the original variable.
   ```
   void increment(int x) { x++; } // x is a copy
   ```
2. Pass by Reference – The function operates on the original variable.
   ```
   void increment(int& x) { x++; }
   // modifies original variable
   ```
3. Pass by Pointer – Similar to reference, but using pointers.
   ```
   void increment(int* x) { (*x)++; }
   ```

4. Return Types

- A function can return a value using the return statement.
- If a function does not return a value, its return type is void.
- Since C++11, you can use auto for return type deduction (especially useful with lambdas or templates).

Example:

```
auto multiply(double a, double b) {
    return a * b;
    // return type deduced as double
}
```

5. Default Arguments

You can specify default values for parameters in the function declaration:

```
void greet(std::string name = "Guest") {
    std::cout << "Hello, " << name << "!\n";
}

greet();          // prints "Hello, Guest!"
greet("Alice"); // prints "Hello, Alice!"
```

Rule: Default arguments must appear after non-default ones.

6. Function Overloading

C++ allows multiple functions with the same name but different parameter lists (number, type, or order of parameters).

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

The compiler selects the correct version based on the arguments provided.

Note: Overloading is not possible based only on return type.

7. Inline Functions

The `inline` keyword suggests to the compiler to insert the function's code directly at the call site (to reduce function call overhead).

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

Modern compilers often ignore `inline` and decide optimization themselves.

8. Recursion

A function can call itself — this is called recursion.

Example: factorial function

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

Be cautious: infinite recursion leads to stack overflow.

9. Best Practices

- Keep functions short and focused on a single task.
- Use meaningful names (calculateTax, not func1).
- Prefer pass-by-reference-to-const for large objects to avoid copying
  `void print(const std::vector<int>& vec);`
- Always consider error handling (e.g., invalid inputs).

Function Templates in C++

1. Introduction to Generic Programming

C++ supports generic programming—writing code that works with any data type without duplicating logic.

The primary mechanism for this is templates.

A function template allows you to define a single function that can operate on multiple types (e.g., int, double, std::string) while maintaining type safety.

💡 Goal: Write once, use with many types — safely and efficiently.

2. Why Use Function Templates?

Without templates, you might write:

```
int max(int a, int b) { return (a > b) ? a : b; }
double max(double a, double b) { return (a > b) ? a : b; }
std::string max(const std::string& a, const std::string& b)
{ return (a > b) ? a : b; }
```

This is **redundant** and **error-prone**.

With a **function template**, you write it **once**:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Now it works for any type T that supports the > operator.

3. Syntax of Function Templates

Basic Form:

```
template <typename T>
return_type function_name(parameter_list) {
    // function body
}
```

template <typename T> declares a template parameter T.

You can also use class instead of typename (they are equivalent here):

```
template <class T>
```

3. Syntax of Function Templates

Example:

```
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(3, 4)         << '\n';  // int
    std::cout << add(2.5, 1.3)     << '\n';  // double
    std::cout << add("Hello, ", "World!"); // ✕ Error! Can't add const char*
}
```

⚠ The compiler instantiates a version of the function for each type used. If the operation isn't valid for a type, compilation fails.

4. Template Type Deduction

You usually don’t need to specify the type explicitly:

```
auto result = add(10, 20);  // T deduced as int
```

But you can specify it if needed:

```
auto result = add<double>(10, 20);
// forces T = double → returns 30.0
```

This is useful when arguments don’t clearly indicate the type or when you want to override deduction.

5. Multiple Template Parameters

You can have more than one type parameter:

```
template <typename T, typename U>
void printPair(T first, U second) {
    std::cout << "First: " << first <<
    ", Second: " << second << '\n';
}

// Usage:
printPair(42, 3.14);       // T = int, U = double
printPair("Key", "Value");// T = const char*, U = const char*
```

Note: T and U can be the same or different types.

6. Non-Type Template Parameters (Brief Overview)

While less common in function templates, C++ also allows non-type parameters (e.g., integers, pointers known at compile time):

```
template <int N>
void printSize() {
    std::cout << "Array size: " << N << '\n';
}

printSize<10>(); // prints "Array size: 10"
```

However, non-type parameters are more frequently used with class templates.

7. Common Pitfalls

No automatic type conversion between arguments

```
template <typename T>
T add(T a, T b);

add(1, 2.5); // ✕ Error! T cannot be both int and double
```

Fix: Make types match explicitly:

```
add<double>(1, 2.5); // OK
```

Header-only usage

Function templates must be defined in headers (not .cpp files), because the compiler needs the full definition to instantiate them for each type.

Operator/operation must be defined

Your template assumes certain operations exist (e.g., +, ==). If not, compilation fails.

Vectors in C++

Introduction to std::vector

In C++, the std::vector is a dynamic array container provided by the Standard Template Library (STL). It is part of the <vector> header and offers a flexible, efficient, and safe way to store sequences of elements.

Unlike built-in arrays:

- Vectors can grow or shrink at runtime.
- They manage their own memory automatically.
- They provide bounds-checked access (via .at()) and rich member functions.

✅ Key advantage: Combines the speed of arrays with the flexibility of dynamic resizing.

2. Basic Syntax and Declaration

To use std::vector, include the header:

```
#include <vector>
```

General form:

```
std::vector<type> name;        // empty vector
std::vector<type> name(size);
// vector with 'size' default-initialized elements
std::vector<type> name(size, value);// vector with 'size' copies of 'value'
std::vector<type> name{init_list};  // initializer list (C++11+)
```

Examples:

```
std::vector<int> numbers;                      // empty
std::vector<double> scores(10);                // 10 zeros
std::vector<std::string> words(5, "hello");    // 5 copies of "hello"
std::vector<char> letters{'a', 'b', 'c'};      // initializer list
```

3. Common Member Functions

Operation	Function	Description
Add element at end	.push_back(x)	Appends x
Remove last element	.pop_back()	Removes (no return)
Access element	vec[i] or .at(i)	[] is fast; .at(i) throws std::out_of_range on invalid index
Current size	.size()	Number of elements
Capacity	.capacity()	Allocated storage size
Reserve memory	.reserve(n)	Pre-allocate memory for efficiency
Resize	.resize(n)	Change size (adds default or specified values)
Check if empty	.empty()	Returns true if size is 0
Clear all	.clear()	Removes all elements

3. Common Member Functions

Example:

```
std::vector<int> v;
v.push_back(10);
v.push_back(20);
std::cout << v[0] << '\n';          // 10
std::cout << v.size() << '\n';      // 2
v.pop_back();                       // now size = 1
```

4. Iterators and Range-Based Loops

Vectors support iterators, enabling generic algorithms:

```
// Using iterators
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << ' ';
}

// Range-based for loop (C++11+)
for (const auto& elem : v) {
    std::cout << elem << ' ‘;
}
```

🔍 Prefer const auto& to avoid unnecessary copying, especially for large objects.

5. Iterating Over Vector Elements in C++

In C++, there are several common and idiomatic ways to loop through (or iterate over) the elements of a std::vector. Each method has its own use cases and advantages.

1. Range-Based for Loop (C++11 and later)

✅ Most common and recommended for simple traversal

```
#include <vector>
#include <iostream>
std::vector<int> vec = {10, 20, 30};
// Read-only access
for (const auto& element : vec) {
    std::cout << element << ' ';
}
// Modify elements
for (auto& element : vec) {
    element *= 2;
}
```

5. Iterating Over Vector Elements in C++

- Use const auto& to avoid copying (especially important for large or non-trivial types).
- Use auto& if you need to modify the elements.
- Clean, safe, and concise.

5. Iterating Over Vector Elements in C++

2. Traditional for Loop with Index

```
for (size_t i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << ' ‘;
}
```

- Gives you direct access to the index, which is useful when you need positional information.
- ⚠ Be cautious: vec.size() returns size_t (unsigned). Avoid decrementing below zero.

🔒 Safer alternative with signed index (C++20):

```
for (int i = 0;
i < static_cast<int>(vec.size()); ++i)
```

5. Memory Management & Performance

- Vectors store elements in contiguous memory (like arrays), enabling cache-friendly access.
- When capacity is exceeded, the vector reallocates (typically doubling capacity), which involves copying/moving elements.
- To avoid frequent reallocations:
  - Use .reserve(n) if you know the approximate number of elements.
  - Use .shrink_to_fit() to reduce excess capacity (non-binding request).

  Example:

```
std::vector<int> v;
v.reserve(1000);
// avoids reallocations for first 1000 push_backs
```