Inline Function in C

Pedram Pasandide

Introduction

An **inline function** in C is a function that the compiler attempts to expand **in place** rather than performing a standard function call. This can **reduce function call overhead** and improve performance, especially for small, frequently used functions.

Why Use Inline Functions?

1. **Eliminates Function Call Overhead**: Every function call involves pushing arguments onto the stack, jumping to the function, executing, and returning. Inline functions avoid this overhead.
2. **Improves Performance**: Useful for small, frequently called functions like mathematical operations or accessor functions.
3. **Enables Compiler Optimizations**: The compiler may optimize inline functions more aggressively than normal functions.
4. **Better Code Readability & Maintainability**: Unlike macros (`#define`), inline functions provide type safety and debugging benefits.

In C99 and later, you can declare an inline function using the inline keyword:

```
inline return_type function_name(parameters) {
 // Function body
}
```

The compiler **may or may not** actually inline the function based on optimization settings and function complexity.

Let's consider a simple example:

```
#include <stdio.h>

// Declaring an inline function
inline int square(int x) {
 return x * x;
}

int main() {
 int num = 5;
 printf("Square of %d is %d\n", num, square(num));
 return 0;
}
```

The function `square()` is marked as `inline`, so wherever `square(num)` appears, the compiler may **replace it** with `num * num` instead of making a function call. This avoids the overhead of calling a function and returning the result.

Before inline functions, macros (`#define`) were used for small computations:

```
#define SQUARE(x) (x * x)
```

However, macros have some issues:

1. **Lack of Type Safety**: If `x` is a floating-point number, a macro may not behave as expected.
2. **Multiple Evaluations**: Consider `SQUARE(5 + 1)`, which expands to `(5 + 1 * 5 + 1)`, leading to unexpected results.

Inline functions solve these issues by being type-safe and behaving like regular functions.

Even when a function is declared `inline`, the compiler may ignore the inline request based on:

1. **Function Complexity**: Large or complex functions may not be inlined.
2. **Compiler Optimization Settings**: `-O2` or `-O3` optimizations in GCC can influence inlining.

To enforce inlining in GCC/Clang, use:

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

Use Inline Functions when:

- Small utility functions (e.g., math operations, getters/setters).
- Frequently called functions in performance-critical code.

Avoid using Inline when

- Large functions (increases code size, reduces cache efficiency).
- Recursive functions (cannot be inlined).

The following table show a summary of what we discuss:

Table 1: A comparison between `inline` and `#define` functions

Feature	Inline Function	Macro (`#define`)
Type-Safe	Yes	No
Debugging Support	Yes	No
Function Overhead	No (if inlined)	No
Readability	Better	Worse

Inline functions provide a great alternative to macros by offering type safety, better debugging, and function-like behavior while still optimizing performance.