inline function specifier

Declares an inline function .

Syntax

`inline` *function_declaration* (since C99)

Explanation

The intent of the **inline** specifier is to serve as a hint for the compiler to perform optimizations, such as function inlining , which usually require the definition of a function to be visible at the call site. The compilers can (and usually do) ignore presence or absence of the `inline` specifier for the purpose of optimization.

If the compiler performs function inlining, it replaces a call of that function with its body, avoiding the overhead of a function call (placing data on stack and retrieving the result), which may result in a larger executable as the code for the function has to be repeated multiple times. The result is similar to function-like macros, except that identifiers and macros used in the function refer to the definitions visible at the point of definition, not at the point of call.

Regardless of whether inlining takes place, the following semantics of inline functions are guaranteed:

Any function with internal linkage may be declared `static inline` with no other restrictions.

A non-static inline function cannot define a non-const function-local static and cannot refer to a file-scope static.

```
static int x;
inline void f(void)
{
    static int n = 1; // error: non-const static in a non-static inline function
    int k = x; // error: non-static inline function accesses a static variable
}
```

If a non-static function is declared `inline`, then it must be defined in the same translation unit. The inline definition that does not use `extern` is not externally visible and does not prevent other translation units from defining the same function. This makes the **inline** keyword an alternative to `static` for defining functions inside header files, which may be included in multiple translation units of the same program.

If a function is declared `inline` in some translation units, it does not need to be declared `inline` everywhere: at most one translation unit may also provide a regular, non-inline non-static function, or a function declared `extern inline`. This one translation unit is said to provide the *external definition*. In order to avoid undefined behavior, one external definition must exist in the program if the name of the function with external linkage is used in an expression, see one definition rule.

The address of an inline function with external linkage is always the address of the external definition, but when this address is used to make a function call, it's unspecified whether the *inline definition* (if present in the translation unit) or the *external definition* is called. The static objects defined within an inline definition are distinct from the static objects defined within the external definition:

```
inline const char *saddr(void) // the inline definition for use in this file
{
    static const char name[] = "saddr";
    return name;
}
int compare_name(void)
{
    return saddr() == saddr(); // unspecified behavior, one call could be external
}
extern const char *saddr(void); // an external definition is generated, too
```

A C program should not depend on whether the inline version or the external version of a function is called, otherwise the behavior is unspecified.

Keywords

`inline`

Notes

The `inline` keyword was adopted from C++, but in C++, if a function is declared `inline`, it must be declared `inline` in every translation unit, and also every definition of an inline function must be exactly the same (in C, the definitions may be different, and depending on the differences only results in unspecified behavior). On the other hand, C++ allows non-const function-local statics and all function-local statics from different definitions of an inline function are the same in C++ but distinct in C.

Example

Run this code

```
// file test.h
#ifndef TEST_H_INCLUDED
#define TEST_H_INCLUDED
inline int sum (int a, int b)
{
    return a+b;
}
#endif

// file sum.c
#include "test.h"
extern inline int sum (int a, int b); // provides external definition

// file test1.c
#include <stdio.h>
#include "test.h"
extern int f(void);

int main(void)
{
    printf("%d\n", sum(1, 2) + f());
}

// file test2.c
#include "test.h"

int f(void)
{
    return sum(2, 3);
}
```

Output:

```
8
```

References

- C11 standard (ISO/IEC 9899:2011):
    - 6.7.4 Function specifiers (p: 125–127)
- C99 standard (ISO/IEC 9899:1999):
    - 6.7.4 Function specifiers (p: 112–113)

See also

C++ documentation for **inline specifier**