


INLINE FUNCTIONS IN C
AND THEIR LINKAGE

By

0xLeo (github.com/0xleo)

November 20, 2019

DRAFT X.YY

Missing: ...


Contents

1 Inline functions in C

This is a template that includes all packages and styles I need for writing notes.

1.1 Why use them?

Functions in C can be declared as `inline` to *hint* (but not force) the compiler to optimise the speed of the code where they are used. Although many compilers know when to `inline` a function, it's a good practice to declare them in the source code.

Making a function `inline` means that instead of calling it, its body is copied by the compiler to the caller line. This eliminates the overhead of calling a function (creating stack space, arguments and local variables, and jumping to its definition, push variables to stack, pop etc.). It's a good practice for short functions that are called a few times in the code, otherwise it increases the code size (each call, one copy is added to the code).

> **TAKEAWAY 1.1.** `inline` *is nothing but a hint to the compiler to try to replace a function call with its definition code wherever it's called.*

It may seem that inline functions are similar to macros. They are, but there are two key differences:

- Macros are expanded by the preprocessor before compilation and they *always* substitute the caller text with the body text.
- `inline` functions are type-checked but macros are not since macros are just text.

Let's create an inline function, call it and see what happens. If we try to compile the code below (without optimisations), i.e.

```
gcc inline_error.c -o inline_error
```

we get the linker error:

Listing 1: Attempting to declare an inline function (src/inline_error.c).

```
inline int foo()
{
    return 0xaa;
}

int main(int argc, char *argv[])
{
    int ret;

    ret = foo();
    return 0;
}
```

```
inline_error.c:(.text+0x12): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

In this case, the compiler has chosen *not to* inline `foo`, searches its definition symbol and cannot find it. However, if we compile with optimisations, i.e.

```
gcc -O inline_error.c -o inline_error
```

, then everything will work. `foo` will be inlined and so the linker will not need the "regular" definition.

1.2 Linkage issues – `static inline` vs `extern inline`

The C ISO, section 6.7.4[1], defines the following regarding the linkage of inline functions.

> *Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply:*
>
> *If a function is declared with an inline function specifier, then it shall also be defined in the same translation unit.*

[1] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

If all of the file scope declarations for a function in a translation unit (TU) [2] include the inline function specifier without `extern`, then the definition in that TU is an inline definition. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another TU. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same TU. It is unspecified whether a call to the function uses the inline definition or the external definition. ❞

When we inline a function, we want to have both the "regular" and inline def's available.

The key part of this specification is "an inline definition does not provide an external definition for the function, and does not forbid an external definition in another TU." When we defined function `foo()` in Listing 2 and used it, although it was in the same file, the call to that function is resolved by the linker not the compiler, because it is implicitly `extern`. But this definition alone does not provide an external definition of the function. That's how inline functions differ from regular ones. Also, note the part "it is unspecified whether a call to the function uses the inline definition or the external definition.". That means that if let's say, we have defined a function `inline int foo()`. When the function is called, it's up to the compiler to choose whether to inline it or not. If it chooses not to, it will call `int foo()`. But the symbol for `int foo()` is not defined, hence the error. If it chooses to inline it, it will of course find it and link it so no error.

`inline` functions in `gcc` should be declared `static` or `extern`.

> **TAKEAWAY 1.2.** *The definition of an inline function must be present in the TU where it is accessed.*

To resolve the missing definition behaviour it is recommended that linkage always be resolved by declaring them as `static inline` or `extern inline`. Which one is preferred though?

1.2.1 `static inline`

If the function is declared to be a `static inline` then, as before the compiler may choose to inline the function. In addition the compiler may emit a locally scoped version of the function in the object file if necessary. There can be one static version per object file, so you may end up with multiple definitions of the same function, so if the function is long this can be very space inefficient. The reason for this is that the compiler will generate the function definition (body) in every TU that calls the inline function.

`static inline` means "We have to have this function. If you use it but don't inline it then make a static version of it in this TU." – Linus

> **TAKEAWAY 1.3.** `static` *means "compile the function only with the current TU and then link it only with it".*

Listing 2 demonstrates a case where both the "regular" and inline definitions are needed. In this case, the compiler will inline the code to compute `x*x`. However, next, it will search for the address of the (regular) `square` function. In `square` was only inlined, the address would not be found as the definition symbol would not exist.

Listing 2: `static inline` demonstration (src/inline_static.c).

```
#include <stdio.h>

static inline unsigned int square(int x)
{
    return x*x;
}

int main(int argc, char *argv[])
{
    unsigned int i = square(5);
    printf("address = 0x%x, return = 0x%x\n", (unsigned int) &square, i);
    return 0;
}
```

Create the object file with optimisations[3]:

```
gcc -g -O -c inline_static.c -o inline_static.o
```

View the symbol table for the object file:

```
objdump -t -M intel inline_static.o
```

[2] translation unit (TU) = source file after it has been pre-processed - i.e. after all the `#ifdef`, `#define` etc. have been resolved.
[3] Code highlighted in grey indicates it has been entered in the command line.

```
inline_static.o:     file format elf32-i386
SYMBOL TABLE:
00000000 l    df *ABS* 00000000 inline_static.c
00000000 l    d  .text 00000000 .text
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l     F .text 00000008 square
<-- omitted -->
00000008 g     F .text 00000032 main
00000000         *UND* 00000000 __printf_chk
```

Next, observe how both the inlined body and the function call (at `printf`) co-exist in the executable. Create the executable with optimisations:

Functions are more likely to be inlined when the TU is compiled with optimisations.

```
gcc -g -O inline_static.c -o inline_static
```

View the disassembly:

```
gdb -q inline_static
```

```
Reading symbols from inline_static...done.
```

```
(gdb) disas square
```

```
Dump of assembler code for function square:
   0x0804842b <+0>: mov    eax,DWORD PTR [esp+0x4]
   0x0804842f <+4>: imul   eax,eax
   0x08048432 <+7>: ret
End of assembler dump.
```

```
(gdb) print &square
```

```
$1 = (int (*)(int)) 0x804842b <square>
```

```
(gdb) disas main
```

```
<-- omitted -->
   0x08048444 <+17>: push   0x19
   0x08048446 <+19>: push   0x804842b
   0x0804844b <+24>: push   0x80484f0
   0x08048450 <+29>: push   0x1
   0x08048452 <+31>: call   0x8048310 <__printf_chk@plt>
   0x08048457 <+36>: add    esp,0x10
<-- omitted -->
```

To print `5*5=0x19`, the compiler directly pushes it in the stack instead of calling `square`, avoiding all the call overhead. At the same time, the function definition exists in the file since its address (`0x804842b`) had to be printed. The takeaway here is that:

> **TAKEAWAY 1.4.** *The compiler will generate function code for a `static inline` only if its address is used.*

The listing below demonstrates it.

Listing 3: In this case, function code for `square` will bbnot be generated. (src/inline_static_no_code.c).

```
#include <stdio.h>

static inline unsigned int square(int x)
{
    return x*x;
}

int main(int argc, char *argv[])
```

```
{
    unsigned int i = square(5);
    printf("return = 0x%x\n", i);
    return 0;
}
```

If we compile it with optimisations and search for the symbol of the `square` function, nothing will be found. It is used solely as inlined. This can also be confirmed by `gdb`.

```
gcc -c -O -g inline_static_no_code.c -o inline_static_no_code.o
objdump -t -M intel inline_static_no_code.o | grep square
```

> **TAKEAWAY 1.5.** *Short, simple functions are OK to be defined as `static inline` in the TU that calls then as long as they don't generate too much bloat.*

1.3 `extern inline`

In C, all functions are `extern` by default, i.e. visible to other TUs, so for regular functions there's no need to use it.

Declaring a function as `extern` tells the compiler that the storage for this function is defined somewhere and if you haven't seen it's definition that's OK – it will be connected with the linker. Thus extends the visibility of a function (or variable). It is useful when an `inline` function is defined in a header. Then it can be declared `extern` in the `.c` file that wants to call it. The linker will link it to the one in the header and depending on whether the compiler has decided to optimise or not, it will use either the function call or the inline code.

Since the declaration can be done any number of times and definition can be done only once, we notice that declaration of a function can be added in several TUs. But the definition only exists in one TU and it might contain. And as the extern extends the visibility to the whole file, the function with extended visibility can be called anywhere in any TU provided the declaration of the function is known. This way we avoid defining a function with the same body again and again.

A good practice is to declare a function defined somewhere else as `extern`.

> **TAKEAWAY 1.6.** *So the best practice when we want to make an inline function external is:*
>
> ```
> // .h file - regular definition
> void foo(void)
> {
>     ...
> }
>
> // .c caller file - declaration
> extern inline void foo(void);
> ...
> foo();
> ```

EXAMPLE 1.1. We have our simple regular function to inline in a header.

Listing 4: Definition of `foo()` (src/foo.h).

```
#ifndef FOO_H
#define FOO_H

#include <stdio.h>

unsigned int foo(void)
{
    return 0xaa;
}
#endif /* FOO_H */
```

We want our `.c` caller to see it and potentially inline it. As mentioned before, the way to do this is by adding `extern inline` in front of the declaration (Listing 5).

Listing 5: Telling the compiler to use the external def'n of `foo` (src/extern_call_foo.c).

```c
#include <stdio.h>
#include "foo.h"

extern inline unsigned int foo(void);

int main(int argc, char *argv[])
{
    printf("address = 0x%x, ret = 0x%x\n", &foo, foo());
    return 0;
}
```

The following caller, although it does not explicitly declares `foo` as external, would also work since all functions in modern C are external by default. The code produced with or without `extern` is the same. However it's a good practice to use the `extern` keyword to make it clear.

Listing 6: Implicitly telling the compiler to use the external def'n of `foo` (src/extern_call_foo2.c).

```c
#include <stdio.h>
#include "foo.h"

inline unsigned int foo(void);

int main(int argc, char *argv[])
{
    printf("address = 0x%x, ret = 0x%x\n", &foo, foo());
    return 0;
}
```

Both of the last two listings work with or without the `-O` flag - i.e. the compiler is free to choose either the inline or regular version of `foo`. In this case, it will inline `foo()` with `-O`. For the address, it of course needs the full definition.

code	gcc	gcc -O
Listing 5	✔	✔
Listing 6	✔	✔

EXAMPLE 1.2.

In this example, we show how to call the *same* external inline function (address and body) in multiple `.c` files.

When we want to inline a function, its body must be present in the header where it's define As usual, the function we want to inline is `foo`. That's a rare case where we define a function in the header itself as regular functions as declared in `func.h` but defined in `func.c`.

We don't normally define functions in headers but functions we will later declare as `extern inline` are an exception.

Listing 7: Definition of `foo()` that we want to inline (src/foo.h).

```c
#ifndef FOO_H
#define FOO_H

#include <stdio.h>

unsigned int foo(void)
{
    return 0xaa;
}
#endif /* FOO_H */
```

Let's say that we have a function `foo_caller` that marks `foo` as inline, calls it, and prints its address and return declared in `foo_caller.h` and defined in `foo_caller.c`.

Listing 8: Declaration of `foo_caller()` (src/foo_caller.h).

```c
#ifndef FOO_CALLER_H
#define FOO_CALLER_H
```

```
3
4 void foo_caller(void);
5
6 #endif /* FOO_CALLER_H */
```

Listing 9: Declaration of `foo_caller()` (src/foo_caller.c).

```
1 #include <stdio.h>
2 #include "foo_caller.h"
3 #include "foo.h"
4
5 extern inline unsigned int foo(void);
6
7 void foo_caller(void)
8 {
9     printf("foo_caller called foo at 0x%x, ret = 0x%x\n", &foo, foo());
10 }
```

Finally, we have the main function that will mark `foo` as `extern inline` (i.e. tells the compiler its definition is found elsewhere), and call it directly and through `foo_caller`. There's one important detail to note when including `foo`'s header. The header contains its definition. `foo_caller.c` includes `foo.h`, therefore contains one definition of `foo`. `main.c` includes `foo_caller.h`, therefore already contains one definition of `foo`. If we include `foo.h` in `main`, we'll end up with a multiple definition error emitted by the linker. This wouldn't be a problem with regular functions, as they only contain the declaration in the header and a function can be declared infinite times, but it is a problem when a function is defined in the header, e.g. a function we want to inline. In this case, the programmer must manually make sure to include the header only once!

When we define a function in a header, its header must only be included once in the main!

Listing 10: `main` function calling `foo` through two different files (src/main_ext_foo.c).

```
1 #include <stdio.h>
2 #include "foo_caller.h"
3
4 extern inline unsigned int foo(void);
5
6 int main(int argc, char *argv[])
7 {
8     foo_caller();
9     printf("foo called from main at 0x%x, ret = 0x%x\n", &foo, foo());
10     return 0;
11 }
```

This compiles either without or without optimisations and the output is:

```
foo_caller called foo at 0x400526, ret = 0xaa
foo called from main at 0x400526, ret = 0xaa
```

Therefore both `main` and `foo_caller` use the same definition. The compiler is free to choose the inline or regular version of `foo` depending on the optimisation flag. For the address, it will of course always use the regular version as it needs the definition. We can do the usual checks with `gdb` and `objdumb` to confirm the disassembly looks as expected. Some final notes regarding this example.

We have been talking about `extern` functions, but `extern` variables also behave the same way.

1.3.1 Force the GNU C compiler to inline a function

In GNU C, we can force inlining of a function by setting its so-called attribute.

In GNU C (and C++), we can use function attributes to specify certain function properties that may help the compiler optimise calls or check code more carefully for correctness. Function attributes are introduced by the `__attribute__` keyword *in the declaration* of a function, followed by an attribute specification enclosed in double parentheses.

We can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following one attribute specification with another.

To get to the point, the particular attribute to force inlining is `always_inline`. According to `gcc` docs:

> *Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.*

Therefore we can force inlining, e.g. for a `static` function, as follows:

```
static void foo(void)
{
    // ...
}

static inline void foo(void)  __attribute__ ((always_inline));
```

We will experiment with the usual `foo` function:

Listing 11: Force `foo` to be inlined, with or without optimisations (src/force_inline.c).

```
#include <stdio.h>

static unsigned int foo(void)
{
    return 0xaa;
}

static inline unsigned int foo(void) __attribute__ ((always_inline));

int main(int argc, char *argv[])
{
    int i = foo();
    printf("ret = 0x%x\n", i);
    return 0;
}
```

Compiling without optimisations and debugging:

```
gcc -g force_inline.c -o force_inline
gdb force_inline
(gdb) set disassembly-flavor intel
(gdb) disas main
```

We see that the call to `printf`, which prints the return of `foo` is disassembled to the following snippet, which shows that our call has been inlined.

```
0x08048426 <+17>: mov    eax,0xaa
0x0804842b <+22>: mov    DWORD PTR [ebp-0xc],eax
0x0804842e <+25>: sub    esp,0x8
0x08048431 <+28>: push   DWORD PTR [ebp-0xc]
0x08048434 <+31>: push   0x80484d0
0x08048439 <+36>: call   0x80482e0 <printf@plt>
```

1.4 Conclusion – When to use the `inline` keyword?

`static inline` works in both ISO C and GNU C (see A.1), it's natural that people ended up settling for that and seeing that it appeared to work without giving errors. So `static inline` gives portability, although it may result in code bloat.

With the exception of tight loops and trivial functions, inlining is the sort of optimisation that should usually be used only when a performance bottleneck has been discovered through profiling. People suggest that:

- Don't use `inline` unless you know what they do and all of the implications.
- Choosing to use the inline code or not doing carries no guarantees but may improve performance.
- "Premature optimisation is the root of all evil." – D. Knuth.

A Appendices

A.1 ANSI C vs GNU C

In the main text, we have used the terms “ISO C, ANSI C” and “GNU C”. They mean different things.

- GNU C: GNU is a unix like operating system (www.gnu.org) & somewhere GNU’s project needs C programming language based on ANSI C standard. GNU use GCC (GNU Compiler Collection) compiler to compile the code. It has C library function which defines system calls such as malloc, calloc, exit...etc
- ANSI C is a standardised version of the C language. As with all such standards it was intended to promote compatibility between different compilers which tended to treat some things a little differently.
- standard specified in the ANSI X3.159-1989 document became known as ANSI C, but it was soon superseded as it was adopted as an international standard, ISO/IEC 9899:1990.