

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.7.4 Function specifiers

function specifier  
syntax

*function-specifier*:  
**inline**

1522

### Commentary

Rationale The **inline** keyword, adapted from C++ . . .

The keyword **inline** is invariably used by languages to specify functions that are to be considered for inlining by a translator.

#### C90

Support for *function-specifier* is new in C99.

#### C++

The C++ Standard also includes, 7.1.2p1, the *function-specifiers* **virtual** and **explicit**.

#### Other Languages

Few languages specify support for function inlining, although their implementations may provide it as an extension (e.g., some Fortran implementations support some form of **inline** command line option). Some implementations support automatic inlining as an optimization phrase. CHILL supports the **inline** keyword. Ada defines a pragma directive that may be used to pass inlining information to the translator, e.g., `pragma inline(fahr)`.

#### Common Implementations

gcc supported **inline** as an extension to C90.

### Constraints

---

Function specifiers shall be used only in the declaration of an identifier for a function.

1523

#### Commentary

The concept denoted by the only available function specifier has no interpretation for object or incomplete types.

#### Coding Guidelines

Those coding guideline documents that argue against the use of the **register** storage-class specifier may well argue against the use of function specifiers for the same reasons. These coding guidelines do not recommend against this usage for the same reason they did not recommend against the use of the **register** storage-class specifier.

---

An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage.

1524

#### Commentary

Note: this constraint applies to an *inline definition*. Having a function declared using the **inline** specifier is not enough for this constraint to apply.

Functions declared with external linkage and the inline specifier can have more than one definition. Two function definitions containing any of the identifiers described by this constraint could not be called interchangeably. For instance, given the translation unit:

**inline**<sup>1529</sup>  
suggests fast calls

**register**  
extent effective

inline  
static storage  
duration

**inline def-**<sup>1540</sup>  
**inition**

**inline def-**<sup>1541</sup>  
**inition**  
not an external  
definition

```

_____ file_1.c _____
1  #include "f.h"
2
3  void g_1(void)
4  {
5  f();
6  }

```

and another translation unit:

```

_____ file_2.c _____
1  #include "f.h"
2
3  void g_2(void)
4  {
5  f();
6  }

```

and the header file:

```

_____ f.h _____
1  inline void f(void)
2  {
3  static int total;
4
5  total++;
6  }

```

the call to the function `f` within `file_1.c` could result in the inline function defined in that source file being invoked, while the call in `file_2.c` could result in the inline function defined in that source file being invoked. C has a very loose model for handling translation of separate source files, a translator is not required to have knowledge of the contents of `file_1.c` when it is translating `file_2.c` and vice versa. In the above case storage for two copies of `total` will be created and independently referenced. This is likely to be surprising to developers who carefully ensure that only one definition of `f` exists. [linkage](#)

The requirements in this constraint prevent developers being surprised by this case and do not require any separate source file translation techniques beyond those currently required of a translator.

An object defined to be a `const`-qualified type is not modifiable and hence the following is permitted: [modifiable lvalue](#)

```

_____ f.h _____
1  inline void f(void)
2  {
3  static const int magic = 42;
4  }

```

Therefore, the following example might not behave as expected.

Rationale

```

inline const char *saddr(void)
{
    static const char name[] = "saddr";
    return name;
}

int compare_name(void)
{
    return saddr() == saddr(); // unspecified behavior
}

```

Since the implementation might use the inline definition for one of the calls to `saddr` and use the external definition for the other, the equality operation is not guaranteed to evaluate to 1 (true). This shows that static objects defined within the inline definition are distinct from their corresponding object in the external definition. This motivated the constraint against even defining a non-`const` object of this type.

## C++

The C++ Standard does not contain an equivalent prohibition.

7.1.2p4 A **static** local variable in an **extern inline** function always refers to the same object.

The C++ Standard does not specify any requirements involving a static local variable in a static inline function. Source developed using a C++ translator may contain inline function definitions that would cause a constraint violation if processed by a C translator.

### Coding Guidelines

The guideline recommendation dealing with identifiers at file scope that are referenced in more than one source file is applicable here.

identifier ??  
declared in one file

---

In a hosted environment, the **inline** function specifier shall not appear in a declaration of `main`.

1525

### Commentary

An implementation may associate special properties with the function called on program startup. In a hosted environment the name of the function called at program startup is known (i.e., `main`). While in a freestanding environment the name of this function is not known (it is implementation-defined).

## C++

hosted environment  
startup  
freestanding environment  
startup

3.6.1p3 A program that declares `main` to be **inline** or **static** is ill-formed.

A program, in a freestanding environment, which includes a declaration of the function `main` (which need not exist in such an environment) using the **inline** function specifier will result in a diagnostic being issued by a C++ translator.

### Semantics

---

A function declared with an **inline** function specifier is an *inline function*.

1526

### Commentary

This defines the term *inline function*, which is a commonly used term, by developers, in many languages.

---

The function specifier may appear more than once;

1527

### Commentary

This permission is consistent with that given for type qualifiers.

## C++

The C++ Standard does not explicitly specify this case (which is supported by its syntax).

### Coding Guidelines

The coding guideline issues for function specifiers occurring more than once are different from those for type qualifiers. There is a single function specifier and the context in which it occurs creates few rationales for the need of more than one to appear. However, while multiple function specifiers might be considered unusual there does not appear to be any significant benefit in a guideline recommendation against this usage.

function specifier  
appears more  
than once

qualifier  
appears more  
than once

qualifier  
appears more  
than once

1528 the behavior is the same as if it appeared only once.

### Commentary

This is consistent with type qualifiers.

1529 Making a function an inline function suggests that calls to the function be as fast as possible.<sup>[18]</sup>

### Commentary

The **inline** specifier is a hint from the developer to the translator. The 1970s gave us the **register** keyword and the 1990s have given us the **inline** keyword. In both eras commercially usable translation technology was not up to automating the necessary functionality and assistance from developers was the solution adopted. Published figures on translators that automatically decide which functions to inline<sup>[3,8,10]</sup> show total program execution time reductions of around 2% to 8%. The latest research on inlining<sup>[18]</sup> has not significantly improved on these program execution time reductions, but it does not seem to cause the large increase in executable program size seen in earlier work, and the translation overhead associated with deducing what to inline has been reduced.

The complexity of selecting which functions to inline, to minimize a program's execution time while keeping the size of the program image below some maximum limit and the size of individual functions below some maximum limit, is known to be at least NP-complete.<sup>[16]</sup>

Developers often believe that function calls take a long time to execute (relative to other instructions). While this might have been true 20 years ago, it is rarely true today. Processor designers have invested significant resources in speeding up the calling of functions (call instructions have been simplified, no complicated VAX type instructions,<sup>[6]</sup> and branch prediction is applied to the call/return instructions, helping to minimize stalls of the instruction pipeline). See Davidson<sup>[8]</sup> for a performance comparison of the effect of **inline** across four different processors.

On modern processors the time taken to execute a call or return instruction is usually less than that required to execute a multiply (although it may slow the execution of other instructions because of a stalled pipeline or failed branch prediction), it can even be as fast as an add instruction. The relative unimportance of call/return instruction performance is shown both by situations where a dramatic decrease in execution time has little impact on overall program performance<sup>[17]</sup> and the fact that putting duplicate sections of code in a translator created function is treated as a potentially worthwhile speed optimization.<sup>[13]</sup> The impact of using separate functions, rather than inlined code, is in areas other than the call/return instruction execution overhead. They include the instruction cache, register usage, and the characteristics of memory (locals on the stack) accesses.

It often comes as a surprise to developers that use of the **register** storage class can slow a program down. The same is also true of the **inline** function specifier; its use can slow a program down (although the situations where this occurs appear to be less frequent than for the **register** storage class). Degradations in performance due to an increase in page swapping (on hosts with limited storage) or an increase in program size causing a decrease in the number of cache hits are the most commonly seen reasons. One published report<sup>[5]</sup> (Fortran source) found that a lack of sophisticated dependency analysis in the translator meant that it had to make worst-case assumptions in a critical loop that did not apply in the non-inlined source. Even when **inline** is used intelligently (based on execution counts of function calls) improvements in performance can vary significantly, depending on the characteristics of the source code and on the architecture of the host processor.<sup>[8,10]</sup>

### C++

The C++ Standard gives an implementation technique, not a suggestion of intent:

*The **inline** specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism.*

7.1.2p2

inline  
suggests  
fast calls

program  
image

processor  
pipeline

branch pre-  
diction

register  
storage-class

Such wording does not prevent C++ implementors interpreting the function specifier in the C Standard sense (by, for instance, giving instructions to the hardware memory manager to preferentially keep a function's translated machine code in cache).

### Common Implementations

register  
storage-class

A parallel can be drawn with the hint provided by the **register** storage class. To what extent will an implementation unconditionally follow the suggestion provided by the appearance of the **inline** function specifier on a function definition, ignore it completely (performing its own analysis of program behavior to deduce which function calls should be inlined<sup>[3]</sup>), or implement some half-way point? Support for this keyword is new in C99 and there are still too few implementations supporting it to be able to spot any trends.

Although inlining is thought of in terms of speeding up the call itself, removing the machine instruction that performs the call is often the smallest saving made. Other savings are obtained from the removal of the interfacing machine code that saves and restores registers across the call, and the code for creating a new stack frame and restoring the old one on return (Davidson<sup>[8]</sup> gives equations for making various cost/benefit decisions and compares predicted behavior against results obtained from four different processors).

Inlining can also have effects beyond the immediate point of call. Many translators treat a single function as the unit of optimization, making worst-case assumptions about the effects of any function calls. Inlining a function allows the statements it contains to be optimized in the context of the call site (arguments are often constant expressions and their values can often replace parameter object accesses in the function body) and also allows information that previously had to be thrown away, because of the call, to be kept and used (a recent innovation, known as *cloning*, changes calls to a function with a call to a copy of that function that has been optimized, based on knowledge of the arguments passed;<sup>[2,18]</sup> when two or more calls to a function share some argument values this technique can provide almost the same performance improvement without the overhead of excessive code expansion).

Inlining a function at the point of call can have disadvantages (and potentially no advantages), including the following:

- The quantity of generated code can increase significantly. Storage to hold generated code is rarely a problem on hosted implementation, but in freestanding implementations it can be a major issue. The increase in size of a program image can also affect the performance of processors instruction cache; the possible effects are complex, depending on size and configuration of the cache.<sup>[4,14]</sup> The Texas Instruments TMS320C compiler<sup>[12]</sup> supports the `-io size` option. The optimizer multiplies the number of times a function is called by its size (an internal, somewhat arbitrary, measure is used) and only inlines the function if the result is less than the developer specified value of `size`.
- The maximum amount of stack storage required by a program can increase. When a function is inlined its stack storage requirements are added to those of the function into which it is merged. Optimizers choose not to inline functions at some call sites if the increase in stack storage requirements exceeds some predefined limit.<sup>[3]</sup>

```

1  inline void f(void)
2  {
3  int af[100];
4  /* ... */
5  }
6
7  void g(void)
8  {
9  int ag[100];
10 /* ... */
11 }
12
13 void h(void)
14 {
15 /*
```

```

16 * When f is not inlined the storage it uses is freed before
17 * the call to g, and so can be reused.
18 * When f is inlined into h the storage it uses becomes part of
19 * the storage allocated to h, and additional storage is required
20 * by the call to g.
21 */
22 f();
23 g();
24 }

```

Inlining not only offers opportunities for reducing the amount of generated code, but also reducing the total amount of stack storage required by nested function calls. Objects with automatic storage duration need to be allocated storage whether the function that defines them is inlined or not. When the function is inlined the storage is allocated in the stack frame of the function into which they are inlined. Inlining thus changes the stack usage profile of a program. Storage requirements can either increase or decrease, depending on what functions are inlined, the housekeeping overhead of a function call and the extent to which it is possible for objects to share storage locations.

Ratliff<sup>[15]</sup> modified `vpcc`<sup>[9]</sup> to attempt to minimize the amount of stack frame storage required for locally defined objects (by using the same storage for different objects, based on the regions of code over which those objects were accessed; the SPARC architecture was used, which has alignment requirements). Table 1529.1 shows the affect of inlining on the amount of storage that is saved.

**Table 1529.1:** Number of bytes of stack space needed by various programs before and after inlining (automatically performed by `vpcc`). *Bytes saved* refers to the amount of storage saved by optimizing the allocation of locally defined objects. Adapted from Ratliff.<sup>[15]</sup>

Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)	Program	Stack Size	Bytes Saved (%)	Inlined Stack Size	Inlined Bytes Saved (%)
ackerman	312	8 (2.56)	232	8 (3.45)	linpack	1,504	48 (3.19)	3,312	112 ( 3.38)
bubblesort	568	8 (1.41)	136	8 (5.88)	mincost	1,216	0 (—)	192	8 ( 4.17)
cal	384	0 (—)	96	0 (—)	prof	1,584	0 (—)	400	40 (10.00)
cmp	768	0 (—)	192	0 (—)	sdiff	2,536	0 (—)	5,784	16 ( 0.28)
csplit	1,488	0 (—)	728	0 (—)	spline	560	8 (1.43)	200	8 ( 4.00)
ctags	8,144	0 (—)	24,544	88 (0.36)	tr	192	0 (—)	96	0 (—)
dhystone	664	0 (—)	200	8 (4.00)	tsp	3,008	8 (0.27)	2,216	56 ( 2.53)
grep	592	0 (—)	304	0 (—)	whetstone	568	0 (—)	488	296 (60.66)
join	480	0 (—)	96	0 (—)	yacc	4,232	0 (—)	1,360	8 ( 0.59)
lex	9,472	0 (—)	7,208	8 (0.11)	average	1,989	4 (0.47)	2,510	34 ( 5.23)

Automatically inlining all functions can lead to very large program images. While heuristics based on number of calls and function size can reduce code expansion, information on which functions are frequently called during program execution enables a more targeted approach to inlining to be made (see Arnold, Fink, Sarkar, and Sweeney<sup>[11]</sup> for a comparison of inlining performance based on using static and dynamic information, Java based).

The translator for the HP—was DEC—Tru68 platform supports the `__forceinline` storage-class modifier.<sup>[11]</sup> Functions declared using this modifier are unconditionally inlined by the translator.

### Coding Guidelines

The term *type safe macro* (because the types of the arguments are checked) or simply *safe macro* (because the arguments are only evaluated once) are sometimes applied to the use of inline functions.

1530 The extent to which such suggestions are effective is implementation-defined.<sup>119)</sup>

### Commentary

There is no requirement on an implementation to handle calls to a function defined with the `inline` function specifier any differently than calls to a function defined without one. This behavior parallels that for the

register  
extent effective

**register** storage-class specifier.

**C++**

7.1.2p2 *An implementation is not required to perform this inline substitution at the point of call;*

A C++ implementation is not required to document its behavior.

### Coding Guidelines

register  
extent effective

Drawing parallels with the implementation-defined behavior of the **register** storage class would suggest that although this behavior is implementation-defined, no recommendation against its use be given. However, there is an argument for recommending the use of **inline** in some circumstances. Developers sometimes use macros because of a perceived performance advantage. Suggesting that an inline function be used instead may satisfy the perceived need for performance (whether or not the translator used performs any inlining is often not relevant), gaining the actual benefit of argument type checking and a nested scope for any object definitions.

---

Any function with internal linkage can be an inline function.

1531

### Commentary

The C Standard explicitly gives this permission because it goes on to list restrictions on inline functions with external linkage.

**C++**

The C++ Standard does not explicitly give this permission (any function declaration can include the **inline** specifier, but this need not have any effect).

---

For a function with external linkage, the following restrictions apply:

1532

### Commentary

The following restrictions define a model that has differences from the one used by C++.

Rationale Inlining was added to the Standard in such a way that it can be implemented with existing linker technology, and a subset of C99 inlining is compatible with C++.

**C++**

The C++ Standard also has restrictions on inline functions having external linkage. But it does not list them in one paragraph.

A program built from the following source files is conforming C, but is ill-formed C++ (3.2p5).

```

1  inline int f(void)
2  {
3  return 0+0;
4  }
```

File a.c

```

1  int f(void)
2  {
3  return 0;
4  }
```

File b.c

Building a program from sources files that have been translated by different C translators requires that various external interface issues, at the object code level, be compatible. The situation is more complicated when the translated output comes from both a C and a C++ translator. The following is an example of a technique that might be used to handle some inline functions (calling functions across source files translated using C and C++ translators is more complex).

```

_____ x.h _____
1  inline int my_abs(int p)
2  {
3  return (p < 0) ? -p : p;
4  }

```

```

_____ x.c _____
1  #include "x.h"
2
3  extern inline int my_abs(int);

```

The handling of the second declaration of the function `my_abs` in `x.c` differs between C and C++. In C the presence of the **extern** storage-class specifier causes the definition to serve as a non-inline definition. While in C++ the presence of this storage-class specifier is redundant. The final result is to satisfy the requirement for exactly one non-inline definition in C, and to satisfy C++'s one definition rule.

C++  
one definition  
rule

1533 If a function is declared with an **inline** function specifier, then it shall also be defined in the same translation unit.

### Commentary

This is a requirement on the program. It removes the need for linker technology that obtains a definition from some place other than the same translation unit.

### C++

*An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).*

7.1.2p4

The C++ Standard only requires the definition to be given if the function is used. A declaration of an inline function with no associated use does not require a definition. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

### Coding Guidelines

The guideline recommendation dealing with placing the textual declaration of identifiers, having external linkage, in a single source file is applicable here.

?? identifier  
declared in one file

1534 118) By using, for example, an alternative to the usual function call mechanism, such as "inline substitution".

### Commentary

Another possibility would be to load the machine code generated from a function body into faster memory, for instance cache memory.

### Other Languages

Most language specifications do not discuss inline function implementation details.

footnote  
118

### Common Implementations

Most translators perform inlining using some internal representation (which is rarely viewable). The output of a function inliner implemented by Davidson and Holler<sup>[7]</sup> was C source code (with functions having been inlined at the point of call).

---

Inline substitution is not textual substitution, nor does it create a new function.

1535

### Commentary

A number of interpretations of the term *inline substitution* are possible. This wording clarifies that the two listed interpretations (with their associated semantics) are not intended to apply. From the developers point of view the semantic of a function call are the abstract one specified by the standard. The only measurable external changes, caused by the addition of a *function-specifier* to the definition of a function, in a program image should be in its size and execution time performance.

An inline function is processed through the eight phases of translation at their point of definition. The sequence of machine code instructions used to implement a call to that function is an internal detail that is not visible to the developer. A few changes have to be made to the machine code when it is inlined. A **return** statement that contains a value has to be changed so that the value is simply treated like any other expression that occurred at the point of the function call. Any invocation of the `va_*` macros still have to refer to the arguments of the function that originally contained them (which may mean creating a dummy function call stack).

### C++

The C++ Standard does not make this observation.

### Coding Guidelines

Inline functions are too new to know whether developers make either of these incorrect assumptions.

---

Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called;

1536

### Commentary

Any other specification of behavior has the potential to cause a change in the appearance of the **inline** function specifier, in a function definition, to change the semantics of a program. Also the existing translation model uses phases of translation. Macro expansion is performed in translation phase 4, prior to any identifiers being converted to keywords.

### C++

The C++ Standard does not make this observation.

---

and identifiers refer to the declarations in scope where the body occurs.

1537

### Commentary

The identifiers that may be in scope where the body occurs, but not where the call to the function occurs all have file scope. They include typedef names and tag names, as well as objects and other functions.

---

Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

1538

### Commentary

It is a requirement on the implementation that the address of every function, during program execution, be unique (also see the response to DR #079). It is also a requirement on the implementation that pointers to different functions do not compare equal.

operator  
()translation phase  
4function  
external linkage  
denotes same  
identifier  
same if inter-  
nal linkage  
pointers  
compare equal

**C++**

An **inline** function with external linkage shall have the same address in all translation units.

7.1.2p4

There is no equivalent statement, in the C++ Standard, for inline functions having internal linkage.

1539 119) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **inline** declaration.

footnote  
119**Commentary**

A translator that operates in a single pass over the source, which the majority do, does not have access to the body of an inline function until its definition is encountered. Consequently it may decide that any calls prior to the definition are not inlined. Other complications that might cause a translator to not perform inline substitution include the following:

imple-  
mentation  
single pass

- When one or more functions forms a recursive chain it may be difficult to fully inline one function into any of the others.
- The `va_*` macros make assumptions about how the parameters they refer to are laid out in storage. The overhead of ensuring that these layout requirements are met, in any inline function that contains uses of these macros, may be sufficient to prevent inline substitution providing any benefit.

1540 If all of the file scope declarations for a function in a translation unit include the **inline** function specifier without **extern**, then the definition in that translation unit is an *inline definition*.

inline definition

**Commentary**

This defines the term *inline definition*. This specification violates the general principle that it be possible to translate C in a single pass. The presence of the word *all* means that a translator has to have seen all declarations of an identifier before it knows whether it is an inline definition. While an implementation still may choose to perform inline substitution before it has processed all of the source (subject to the as-if rule), some constraint requirements only apply to inline definitions, not external definitions, (an implementation is not prohibited from generating spurious diagnostic messages, but it must still correctly translate the source file).

1544 EXAMPLE  
inline  
imple-  
mentation  
single passas-if rule  
1524 inline  
static storage  
duration**C++**

This term, or an equivalent one, is not defined in the C++ Standard.

The C++ Standard supports the appearance of more than one inline function definition, in a program, having a declaration with **extern**. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

**Coding Guidelines**

If the guideline recommendation specifying a single textual definition of an identifier is followed there will never be more than one declaration for a function, in a translation unit, that include the **inline** function specifier.

?? identifier  
declared in one file**Example**

```
1 inline int f_1(void)      /* May be an inline definition. */
2 {}
3 inline int f_2(void)     /* May be an inline definition. */
```

```

4  {}
5  extern inline int f_3(void) /* Not an inline definition. */
6  {}
7  extern inline int f_1(void); /* No, f_1 is not an inline definition. */
8
9                                /* End-of-File, f_2 is an inline definition. */

```

inline definition  
not an external  
definition

An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. 1541

### Commentary

external  
definition

An inline definition is explicitly specified as not being an external definition. The status of an inline function as an inline definition or an external definition does not affect the suggest it provides to a translator.

An inline definition is intended for use only within the translation unit in which it occurs. Because it is not an external definition, the constraint requirement that only one external definition for an identifier occur in a program does not apply. However, if a function is used within an expression an external definition for it must exist somewhere in the entire program. The absence, in a function declaration, of **inline** or the inclusion of **extern** in a declaration creates such an external definition. Also, because an inline definition does not provide an external definition, another translation unit may have to contain an external definition (to satisfy references from other translation units).

external  
linkage  
exactly one  
external definition

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 translation unit. 1542

### Commentary

The inline definition provides all the information (return type and parameter information) needed to call the external definition. Its body can also be used to perform inline substitution. An inline function might be said to have *ghostly linkage*. It exists if the translator believes in it. Otherwise it does not exist and the external definition is referenced.

### C++

In C++ there are no alternatives, all inline functions are required to be the same.

7.1.2p4 *If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.*

A C program may contain a function, with external linkage, that is declared inline in one translation unit but not be declared inline in another translation unit. When such a program is translated using a C++ translator a diagnostic may be issued.

### Coding Guidelines

The guideline recommendation specifying a single textual definition is applicable here.

identifier ??  
declared in one file

It is unspecified whether a call to the function uses the inline definition or the external definition.<sup>120)</sup> 1543

### Commentary

The C Standard does not require that the sequence of tokens representing an inline definition or external definition, of the same function, be the same. However, the intended implication, to be drawn from the unspecified nature of the choice of the definition used, is that a programs external output shall be the same in both cases (apart from execution time performance). If the definitions of the two functions are such that the external program outputs would not be the same, the behavior is undefined.

unspecified  
behavior

**C++**

In the C++ Standard there are no alternatives. An inline definition is always available and has the same definition:

*An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).*

7.1.2p4

Rationale

Second, the requirement that all definitions of an inline function be “exactly the same” is replaced by the requirement that the behavior of the program should not depend on whether a call is implemented with a visible inline definition, or the external definition, of a function. This allows an inline definition to be specialized for its use within a particular translation unit. For example, the external definition of a library function might include some argument validation that is not needed for calls made from other functions in the same library. These extensions do offer some advantages; and programmers who are concerned about compatibility can simply abide by the stricter C++ rules.

**Common Implementations**

The context in which a call occurs can have as much influence over whether a function is inlined as the definition of the function itself. For instance, are there any non-inlined calls nearby (which would have already prevented any flow analysis from building up much information that could be used at the inlined call), does the size of the function exceed some specified, internal translator, limit (there can be other limits such as the amount of storage required by the declarations in a function)?

**Coding Guidelines**

If the guideline recommendation specifying a single textual definition of an identifier is followed the output of a program will not depend on the function chosen. ?? identifier declared in one file

**Example**

In the following example the two definitions of the function `f` are different. The developer has used the fact that the call in `g` occurs in a context where the test performed in the external definition is known to be true. A simplified version of the definition of `f`, applicable to the call made in `g`, has been created and it is hoped that this will result in inline substitution.

```

----- file_1.c -----
1 void f(void)
2 {
3   if (complicated_test) /* A time consuming test. */
4     do_something;
5   else
6     do_something_else;
7 }

```

```

----- file_2.c -----
1 inline void f(void)
2 {
3   do_something;
4 }
5
6 void g(void)
7 {
8   if (part_of_complicated_test)
9     {
10    some_code;

```

```

11     if (rest_of_complicated_test)
12         f();
13     }
14 }

```

EXAMPLE  
inline

EXAMPLE The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit. 1544

```

inline double fahr(double t)
{
    return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
    return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double);      // creates an external definition

double convert(int is_fahr, double temp)
{
    /* A translator may perform inline substitutions */
    return is_fahr ? cels(temp) : fahr(temp);
}

```

Note that the definition of **fahr** is an external definition because **fahr** is also declared with **extern**, but the definition of **cels** is an inline definition. Because **cels** has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either may be used for the call.

### Commentary

Although **fahr** is an external definition, an implementation may still choose to inline calls to it, from within the definition of **convert**.

### C++

The declaration:

```
extern double fahr(double);      // creates an external definition
```

does not create a reference to an external definition in C++.

---

**Forward references:** function definitions (6.9.1). 1545

footnote  
120

120) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions. 1546

### Commentary

The only objects that can have static storage duration are those that have no linkage and are not modifiable lvalues.

### C++

inline 1524  
static stor-  
age duration

A **static** local variable in an **extern inline** function always refers to the same object. A string literal in an **extern inline** function is the same object in different translation units.

The C++ Standard is silent about the case where the **extern** keyword does not appear in the declaration.

```
1  inline const char *saddr(void)
2  {
3  static const char name[] = "saddr";
4  return name;
5  }
6
7  int compare_name(void)
8  {
9  return saddr() == saddr(); /* may use extern definition in one case and inline in the other */
10                             // They are either the same or the program is
11                             // in violation of 7.1.2p2 (no diagnostic required)
12 }
```

## References

1. M. Arnold, S. J. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Additive Compilation and Optimization (DYNAMO'00)*, pages 52–64. ACM, Jan. 2000.
2. A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.
3. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software-Practice and Experience*, 22(5):349–369, 1992.
4. W. Y. Chen, P. P. Chang, W. mei W. Hwu, and T. M. Conte. The effect of code expansion optimizations on instruction cache design. Technical Report CRHC-91-17, University of Illinois, Urbana-Champaign, 1991.
5. K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, Mar. 1992.
6. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.
7. J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software-Practice and Experience*, 18(8):775–790, 1988.
8. J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, 1992.
9. J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimization. *Software-Practice and Experience*, 19(1):79–97, 1989.
10. P. Deshpande and A. Somani. A study and analysis of function inlining. [www.cs.wisc.edu/~pmd/pmd.html](http://www.cs.wisc.edu/~pmd/pmd.html), 1997.
11. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.
12. T. Instruments. *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*. Texas Instruments, Inc, spru024e edition, Aug. 1999.
13. K. Kunchithapadam and J. R. Larus. Using lightweight procedures to improve instruction cache performance. Technical Report CS-Technical-Report 1390, University of Wisconsin-Madison, Jan. 1999.
14. S. McFarling. Procedure merging with instruction caches. Technical Report WRL Research Report 91/5, Digital Western Research Laboratory, Mar. 1991.
15. E. J. Ratliff. Decreasing process memory requirements by overlapping run-time stack data. Thesis (m.s.), Florida State University, College of Arts and Sciences, 1997.
16. R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.
17. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, Dec. 1993.
18. T. P. Way. *Procedure restructuring for ambitious optimization*. PhD thesis, University of Delaware, 2002.