Inline function

In the C and C++ programming languages, an **inline function** is one qualified with the keyword `inline`; this serves two purposes. Firstly, it serves as a compiler directive that suggests (but does not require) that the compiler substitute the body of the function inline by performing inline expansion, i.e. by inserting the function code at the address of each function call, thereby saving the overhead of a function call. In this respect it is analogous to the `register` storage class specifier, which similarly provides an optimization hint.[1] The second purpose of `inline` is to change linkage behavior; the details of this are complicated. This is necessary due to the C/C++ separate compilation + linkage model, specifically because the definition (body) of the function must be duplicated in all translation units where it is used, to allow inlining during *compilation*, which, if the function has external linkage, causes a collision during *linking* (it violates uniqueness of external symbols). C and C++ (and dialects such as GNU C and Visual C++) resolve this in different ways.[1]

Example

An `inline` function can be written in C or C++ like this:

```
static inline void swap(int *m, int *n)
{
  int temp = *m;
  *m = *n;
  *n = temp;
}
```

Then, a statement such as the following:

```
swap(x, y);
```

may be translated into (if the compiler decides to do the inlining, which typically requires optimization to be enabled):

```
int temp = x;
x = y;
y = temp;
```

When implementing a sorting algorithm doing lots of swaps, this can increase the execution speed.

Standard support

C++ and Cyp, but not its predecessors K&R C and Cby, have support for `inline` functions, though with different semantics. In both cases, `inline` does not force inlining; the compiler is free to choose not to inline the function at all, or only in some cases. Different compilers vary in how complex a function they can manage to inline. Mainstream C++ compilers like Microsoft Visual C++ and GCC support an option that lets the compilers automatically inline any suitable function, even those not marked as `inline` functions. However, simply omitting the `inline` keyword to let the compiler make all inlining decisions is not possible, since the linker will then complain about duplicate definitions in different translation units. This is because `inline` not only gives the compiler a hint that the function should be inlined, it also has an effect on whether the compiler will generate a callable out-of-line copy of the function (see storage classes of inline functions).

Nonstandard extensions

GNU C, as part of the dialect gnu89 that it offers, has support for `inline` as an extension to C89. However, the semantics differ from both those of C++ and Cyp. armcc in Cyo mode also offers `inline` as a non-standard extension, with semantics different from gnu89 and Cyp.

Some implementations provide a means by which to force the compiler to inline a function, usually by means of implementation-specific declaration specifiers:

- Microsoft Visual C++: `__forceinline`
- gcc or clang: `__attribute__((always_inline))` or `__attribute__((__always_inline__))`, the latter of which is useful to avoid a conflict with a user-defined macro named `always_inline`.

Indeterminate cases of that result in larger code (bloated executable file), minimal or no performance gain, and in some cases even a loss in performance. Moreover, the compiler cannot inline the function in all circumstances, even when inlining is forced; in this case both gcc and Visual C++ generate warnings.

Forcing inlining is useful if

- `inline` is not respected by the compiler (ignored by compiler cost/benefit analyzer), and
- inlining results in a necessary performance boost

For code portability, the following preprocessor directives may be used:

```
#ifdef _MSC_VER
  #define forceinline __forceinline
#elif defined(__GNUC__)
  #define forceinline inline __attribute__((__always_inline__))
#elif defined(__CLANG__)
  #if __has_attribute(__always_inline__)
    #define forceinline inline __attribute__((__always_inline__))
  #endif
#else
  #define forceinline inline
#endif
```

Storage classes of inline functions

`static inline` has the same effects in all C dialects and C++. It will emit a locally visible (out-of-line copy of the) function if required.

Regardless of the storage class, the compiler can ignore the `inline` qualifier and generate a function call in all C dialects and C++.

The effect of the storage class `extern` when applied or not applied to `inline` functions differs between the C dialects[2] and C++.[3].

C99

In Cyo, a function defined `inline` will never, and a function defined `extern inline` will always, emit an externally visible function. Unlike in C++, there is no way to ask for an externally visible function shared among translation units to be emitted only if required.

If `inline` declarations are mixed with `extern inline` declarations or with unqualified declarations (ie., without `inline` qualifier or storage class), the translation unit must contain a definition (no matter whether unqualified, `inline`, or `extern inline`) and an externally visible function will be emitted for it.

A function defined `inline` requires exactly one function with that name somewhere else in the program which is either defined `extern inline` or without qualifier. If more than one such definition is provided in the whole program, the linker will complain about duplicate symbols. If, however, it is lacking, the linker does not necessarily complain, because, if all uses could be inlined, it is not needed. But it may complain, since the compiler can always ignore the `inline` qualifier and generate calls to the function instead, as typically happens if the code is compiled without optimization. (This may be the desired behavior, if the function is supposed to be inlined everywhere by all means, and an error should be generated if it is not.) A convenient way is to define the `inline` functions in header files and create one *.c* file per function, containing an `extern inline` declaration for it and including the respective header file with the definition. It does not matter whether the declaration is before or after the include.

To prevent unreachable code from being added to the final executable if all uses of a function were inlined, it is advised[2] to put the object files of all such *.c* files with a single `extern inline` function into a static library file, typically with `ar rcs`, then link against that library instead of the individual object files. That causes only those object files to be linked that are actually needed, in contrast to linking the object files directly, which causes them to be always included in the executable. However, the linking order must be specified after all the other static library files on the linker command line, since calls from object files specified after the library file to the functions will not be considered by the linker. Calls from `inline` functions to other `inline` functions will be resolved by the linker automatically (the `-s` option in `ar rcs` ensures this).

An alternative solution is to use link time optimization instead of a library. gcc provides the flag `-Wl,--gc-sections` to omit sections in which all functions are unused. This will be the case for object files containing the code of a single unused `extern inline` function. However, it also removes any and all other unused sections from all other object files, not just those related to unused `extern inline` functions. (It may be desired to link functions into the executable that are to be called by the programmer from the debugger rather than by the program itself, eg., for examining the internal state of the program.) With this approach, it is also possible to use a single *.c* file with all `extern inline` functions instead of one *.c* file per function. Then the file has to be compiled with `-fdata-sections -ffunction-sections`. However, the gcc manual page warns about that, saying "Only use these options when there are significant benefits from doing so."

Some recommend an entirely different approach, which is to define functions as `static inline` instead of `inline` in header files[2]. Then, no unreachable code will be generated. However, this approach has a drawback in the opposite case: Duplicate code will be generated if the function could not be inlined in more than one translation unit. The emitted function code cannot be shared among translation units because it must have different addresses. This is another drawback; taking the address of such a function defined as `static inline` in a header file will yield different values in different translation units. Therefore, `static inline` functions should only be used if they are used in only one translation unit, which means that they should only go to the respective *.c* file, not to a header file.

gnu89

gnu89 semantics of `inline` and `extern inline` are essentially the exact opposite of those in Cyp[4], with the exception that gnu89 permits redefinition of an `extern inline` function as an unqualified function, while Cyp `inline` does not[5]. Thus, a function without redefinitions is like Cyp `inline`, and gnu89 `inline` is like Cyp `extern inline`; in other words, in gnu89, a function defined `inline` will always and a function defined `extern inline` will never emit an externally visible function. The rationale for this is that it matches variables, for which storage will never be reserved if defined as `extern` and always if defined without. The rationale for Cyp, in contrast, is that it would be astonishing if using `inline` would have a side-effect—to always emit a non-inlined version of the function—that is contrary to what its name suggests.

The remarks for Cyp about the need to provide exactly one externally visible function instance for inlined functions and about the resulting problem with unreachable code apply mutatis mutandis to gnu89 as well.

gcc up to and including version 4.2 used gnu89 `inline` semantics even when `-std=c99` was explicitly specified.[6] With version *j*[7], gcc switched from gnu89 to the gnu11 dialect, effectively enabling Cyp `inline` semantics by default. To use gnu89 semantics instead, they have to be enabled explicitly, either with `-std=gnu89` or, to only affect inlining, `-fgnu89-inline`, or by adding the `gnu_inline` attribute to all `inline` declarations. To ensure Cyp semantics, either `-std=c99`, `-std=c11`, `-std=gnu99` or `-std=gnu11` (without `-fgnu89-inline`) can be used.[3]

C++

In C++, a function defined `inline` will, if required, emit a function shared among translation units, typically by putting it into the common section of the object file for which it is needed. The function must have the same definition everywhere, always with the `inline` qualifier. In C++, `extern inline` is the same as `inline`. The rationale for the C++ approach is that it is the most convenient way for the programmer, since no special precautions for elimination of unreachable code must be taken and, like for ordinary functions, it makes no difference whether `extern` is specified or not.

The `inline` qualifier is automatically added to a function defined as part of a class definition.

armcc

armcc in Cyo mode provides `extern inline` and `inline` semantics that are the same as in C++: Such definitions will emit a function shared among translation units if required. In Cyo mode, `extern inline` always emits a function, but like in C++, it will be shared among translation units. Thus, the same function can be defined `extern inline` in different translation units[7]. This matches the traditional behavior of Unix C compilers[8] for multiple non-`extern` definitions of uninitialized global variables.

Restrictions

Taking the address of an `inline` function requires code for a non-inlined copy of that function to be emitted in any case.

In Cyo, an `inline` or `extern inline` function must not access `static` global variables or define non-const `static` local variables. `const static` local variables may or may not be different objects in different translation units, depending on whether the function was inlined or whether a call was made. Only `static inline` definitions can reference identifiers with internal linkage without restrictions; those will be different objects in each translation unit. In C++, both `const` and non-`const` `static` locals are allowed and they refer to the same object in all translation units.

gcc cannot inline functions if:[9]

1. they are variadic,
2. use `alloca`
3. use computed `goto`
4. use nonlocal `goto`
5. use nested functions
6. use `setjmp`
7. use `__builtin_longjmp`
8. use `__builtin_return`, or
9. use `__builtin_apply_args`

Based on Microsoft Specifications at MSDN, MS Visual C++ cannot inline (not even with `__forceinline`), if

1. The function or its caller is compiled with /O0 (the default option for debug builds).
2. The function and the caller use different types of exception handling (C++ exception handling in one, structured exception handling in the other).
3. The function has a variable argument list.
4. The function uses inline assembly, unless compiled with /Og, /Ox, /O1, or /O2.
5. The function is recursive and not accompanied by `#pragma inline_recursion(on)`. With the pragma, recursive functions are inlined to a default depth of 16 calls. To reduce the inlining depth, use `inline_depth` pragma.
6. The function is virtual and is called virtually. Direct calls to virtual functions can be inlined.
7. The program takes the address of the function and the call is made via the pointer to the function. Direct calls to functions that have had their address taken can be inlined.
8. The function is also marked with the naked `__declspec` modifier.

Problems

Besides the problems with inline expansion in general, `inline` functions as a language feature may not be as valuable as they appear, for a number of reasons:

- Often, a compiler is in a better position than a human to decide whether a particular function should be inlined. Sometimes the compiler may not be able to inline as many functions as the programmer indicates.
- An important point to note is that the code (of the `inline` function) gets exposed to its client (the calling function).
- As functions evolve, they may become suitable for inlining where they were not before, or no longer suitable for inlining where they were before. While inlining or un-inlining a function is easier than converting to and from macros, it still requires extra maintenance which typically yields relatively little benefit.
- Inline functions used in *proliferation* in native C-based compilation systems can increase compilation time, since the intermediate representation of their bodies is copied into each call site.
- The specification of `inline` in C99 requires exactly one external definition of the function, if it is used somewhere. If such a definition wasn't provided by the programmer, that can easily lead to linker errors. This can happen with optimization turned off, which typically prevents inlining. Adding the definitions, on the other hand, can cause unreachable code if the programmer does not carefully avoid it, by putting them in a library for linking, using link time optimization, or `static inline`.
- In C++, it is necessary to define an `inline` function in every module (translation unit) that uses it, whereas an ordinary function must be defined in only a single module. Otherwise it would not be possible to compile a single module independently of all other modules. Depending on the compiler, this may cause each respective object file to contain a copy of the function's code, for each module with some use that could not be inlined.
- In embedded software, oftentimes certain functions need to be placed in certain code sections by use of special compiler instructions such as "pragma" statements. Sometimes, a function in one memory segment might need to call a function in another memory segment, and if inlining of the called function occurs, then the code of the called function might end up in a segment where it shouldn't be. For example, high-performance memory segments may be very limited in code space, and if a function belonging in such a space calls another large function that is not meant to be in the high-performance section and the called function gets inappropriately inlined, then this might cause the high-performance memory segment to run out of code space. For this reason, sometimes it is necessary to ensure that functions do not become inlined.

Quotes

> "A function declaration [. . .] with an **inline** specifier declares an inline function. 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. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions defined by 7.1.2 shall still be respected."
> — ISO/IEC 14882:2011, the current C++ standard, section 7.1.2
>
> "A function declared with an **inline** function specifier is an inline function. [. . .] Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined (*footnote: For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an inline declaration.*)
>
> "[. . .] An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. 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. It is unspecified whether a call to the function uses the inline definition or the external definition."
> — ISO 9899:1999(E), the C99 standard, section 6.7.4

See also

- Macro (computer science)

References

1. Meyers, Randy (July 1, 2002). "The New C: Inline Functions" (http://www.drdobbs.com/the-new-c-inline-functions/184401540).
2. http://www.greenend.org.uk/rjk/tech/inline.html
3. https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/Inline.html
4. http://blahg.josefsipek.net/?p=529
5. https://gcc.gnu.org/gcc-5/porting_to.html
6. https://gcc.gnu.org/ml/gcc-patches/2007-02/msg00119.html
7. http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka15831.html
8. gcc manual page, description of `-fno-common`.
- JANIL DEBASISH (1 January 2005). *C++ AND OBJECT-ORIENTED PROGRAMMING PARADIGM* (https://books.google.com/books?id=DnsM0WD-6iMC&pg=PA131). PHI Learning Pvt. Ltd. ISBN 978-81-203-2871-6.
- Sengupta, Probal (1 August 2004). *Object-Oriented Programming: Fundamentals And Applications* (https://books.google.com/books?id=ZLzt5WtsbzIC&pg=PA50). PHI Learning Pvt. Ltd. ISBN 978-81-203-1258-6.
- Svenk, Goran (2003). *Object-oriented Programming: Using C++ for Engineering and Technology* (https://books.google.com/books?id=MqT3j_1i4CIC&pg=PA16). Cengage Learning. ISBN 0-7668-3894-3.
- Balagurusamy (2013). *Object Oriented Programming with C++* (https://books.google.com/books?id=WCKOAgAADBAJ&pg=PA74). Tata McGraw-Hill Education. ISBN 978-1-259-02903-6.
- Kirch-Prinz, Ulla; Prinz, Peter (2002). *A Complete Guide to Programming in C++* (https://books.google.com/books?id=-yhuTDWg_QcC&pg=PA181). Jones & Bartlett Learning. ISBN 978-0-7637-1817-6.
- Conger, David (2006). *Creating Games in C++: A Step-by-step Guide* (https://books.google.com/books?id=1F6goq07DcC&pg=PA79). New Riders. ISBN 978-0-7357-1434-2.
- Skinner, M. T. (1992). *The Advanced C++ Book* (https://books.google.com/books?id=fpGLZ7WYvCMC&pg=PA97). Silicon Press. ISBN 978-0-929306-10-0.
- Love (1 September 2005). *Linux Kernel Development* (https://books.google.com/books?id=NXVkcCjPbcC&pg=PA18). Pearson Education. ISBN 978-81-7758-910-8.
- DEHURI, SATCHIDANANDA; JAGADEV, ALOK KUMAR; RATH, AMIYA KUMAR (8 May 2007). *OBJECT-ORIENTED PROGRAMMING USING C++* (https://books.google.com/books?id=fxUVrhJD4kQC&pg=PA78). PHI Learning Pvt. Ltd. ISBN 978-81-203-3085-6.

External links

- Inline functions (https://gcc.gnu.org/onlinedocs/gcc/Inline.html) with the GNU Compiler Collection (GCC)