Inline function in c pdf

I'm not robot

reCAPTCHA

Continue

From cppreference.com declares an inline function. [edit] Syntax inline function_declaration (from C99) [edit] Explanation The intention of the inline specifier is to serve as an aid to the compiler to perform optimizations, such as inlining functions that usually require the function definition to be visible at the call location. Compilers can (and usually do) ignore the presence or absence of an inline specifier for optimization purposes. If the compiler performs the inlining function, it replaces the call of that function with its body to avoid overhead cost of the function call (placing the data on the stack and retrieving the result), which may result in a larger executable file than the code for the function must be repeated multiple times. The result is similar to a macro-like function, except that the identifiers and macros used in the function refer to the definitions visible at the definition point, not at the call location. Regardless of whether inlining takes place, the following semantics of inline functions are guaranteed: Any internal link function can be declared static inline without further limitations. The non-static inline function cannot define non-const functions locally static and cannot refer to a static range file. static int x; inline invalid f(invalid) { static int n = 1; // error: non-const static in non-static inline function int k = x; // error: non-static inline function accesses static variables } If a non-static function is declared inline, then it must be defined in the same translation unit. A nested definition that is not externally used is not externally visible and does not prevent other translators from defining the same function. This makes nested keyword an alternative to static for defining functions inside a file header that can be included in multiple translation units of the same program. If a function is declared inline in some translation units, it may not be declared inline everywhere: at the latest one translation unit may also provide a regular, non-line non-technical or external inline function. This translation unit is said to provide an external definition. To prevent undefined behavior, one external definition must exist in a program if a function name with an external link is used in an expression, see one definition rule. The address of a nested function with an external link is always the address of an external definition, but if the address is used to call a function, it is not known whether it is called a nested definition (if present in a translation unit) or an external definition. Static objects defined in the inline definition are different from static objects defined in the external definition: inline const char *saddr(void) // inline definition for use in this file { static const char name[] = saddr; } int compare_name(void) { return saddr() == saddr(), // unspecified behavior, one call may be external } external const char *saddr(invalid); external definition is generated, too C program should not depend on whether the nested version or external version of the function is called, otherwise the behavior is not specified. [edit] Keywords nested [edit] Comments The nested keyword was received from C++, but in C++, if the function is declared nested, it must be declared nested in each translation unit, and also each definition of the inline function must be exactly the same (in C the definitions may vary and, depending on the differences, only lead to unspecified behavior). On the other hand, C++ allows non-const functions-local statics and all function-local statics from different definitions of inline functions are the same in C++, but different in C. [edit] Example // 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 external inline int sum (int a, int b); provides an external definition of // file test1.c #include &lt;stdio.h&gt; #include test.h external int f(void);   int main(invalid) { printf(%d, sum(1, 2) + f()), }   test2.c file #include test.h int f(invalid) { return amount (2, 3); } Output: [edit] References C11 standard (ISO/IEC 9899:2011): 6.7.4 Function specifier (p: 125-127) C99 standard (ISO/IEC 9 899:1999): 6.7.4 Function Specifier (p: 112-113) C++ documentation for inline specifier This article covers nested functions in C and C++. For inline expansion more generally, see Inline Expansion. This article needs to be updated. The reason given is: The meaning of inline changed in C++ (. Update this article to reflect recent events or newly available information. (April 2019) In programming languages C and C++, nested function is a feature qualified for nested keyword; it serves two purposes. First, it serves as a compiler directive that suggests (but does not require) that the compiler replace the body functions inline by performing inline expansion, i.e. by inserting a functional code to address each function call, thereby saving the overhead cost of the function call. In this regard, it is analogous to the registrar storage class specifier, which similarly provides optimization tip. [1] The second purpose of inline is to change the behaviour of the link; details of this are complex. This is necessary because of the separate compilation of the C/C++ + link model, especially since the definition (body) of the function must be duplicated in all translation units where it is used to allow inlination during compilation, which, if the function has an external connection, causes a collision during the link (violates the uniqueness of the external symbols). C and C++ (and dialects &lt;/stdio.h&gt;how to solve GNU C and Visual C++) in different ways. [1] Example Inline function can be written in C or C++ as follows: inline void swap(int*m, int*n) { int tmp = *m, *m = *n; *n = tmp; } Then a statement such as the following: swap(&amp;x, &amp;y); can be translated into (if the compiler decides to do inlining, which usually requires optimization to be enabled): int tmp = x; x = y; y = tmp; When implementing a sorting algorithm does a lot of swaps, it can increase the execution speed. Standard support for C++ and C99, but not its predecessors K&amp;R C and C89, have support for inline features, although with different semantics. In both cases, inline unarmed inlining; the compiler is free to not send this feature at all, or only in some cases. Different compilers differ in how complex features they can manage inline. Common C++ compilers such as Microsoft Visual C++ and GCC support an option that allows compilers to automatically nest any suitable feature, even those that are not marked as nested features. However, simply omit the nested keyword to let the compiler make all inlining decisions is not possible because 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 feature should be inlined, it also affects whether the compiler will generate a callable out-of-line copy of the feature (see storage classes of inline functions). The non-standard GNU C extension, as part of the gnu89 dialect it offers, has support for inline as an extension to the C89. However, semantics differ from both C++ and C99 semantics. armcc in C90 mode also offers inline as a non-standard extension, with semantics different from gnu89 and C99. Some implementations provide the means by which the compiler can nest a function, usually through 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 conflict with a user-defined macro named always_inline. Indisc contrast to use, that can result in a larger code (inflated executable file), minimal or no performance gain, and in some cases even loss of performance. In addition, the compiler cannot suit the function at all times, even if inlining is forced; in this case, both GCC and Visual C++ generate warnings. Forcing inlining is useful if the compiler does not respect the inline (ignored by the compiler cost-benefit analyzer) and results in #ifdef _MSC_VER #define the necessary performance enhancement The following preprocessor guidelines may be used for code portability: #ifdef _MSC_VER #define forceinline __forceinline #elif defined(__GNUC__) #define forceinline inline inline #elif defined(__CLANG__) #if #if #define forceinline inline __attribute__((__always_inline__)) #else #define forceinline inline #endif #else #define forceinline inline #endif Storage classs of inline functions static inline has the same effects in all C dialects and C++. If necessary, it emits a locally visible (not-of-line copy) function. Regardless of the storage class, the compiler can ignore the inline qualifier and generate function calls in all dialects C and C++. The effect of an external storage class, if applied or not applied to inline functions, varies between dialects C(2) and C++[3]. The C99 V C99 function defined inline never and the function defined by the external inline will always issue an externally visible function. Unlike C++, there is no way to request an externally visible feature shared between translation units to be issued only when needed. If inline declarations are mixed with an external inline declaration or with unqualified declarations (i.e. without an inline qualifier or storage class), the translation unit shall contain a definition (whether unqualified, inline or externally inline) and shall be an externally visible function for it. A function defined inline requires exactly one function with this name somewhere else in the program that is defined either externally inline or without a qualifier. If more than one such definition is provided throughout the program, the linker will complain about duplicate symbols. However, if missing, the linker does not necessarily complain, because if all the uses could be lining, there is no need. But it can complain because the compiler can always ignore the inline qualifier and generate calls to the function instead, as usually happens when the code is compiled without optimization. (This may be the desired behavior if the function is to be inlined everywhere by all means, and an error should be generated if it is not.) A convenient way is to define nested functions in header files and create a single .c file per function that contains an external inline statement for it and including the appropriate header file with a definition. It doesn't matter if the statement is before or after inclusion. To prevent unattainable code from being added to the final executable file if all uses of the feature have been inlaid, it is recommended[3] to put the object files of all such .c files with one external inline function into a static library file, usually with ar rcs, then reference against this library instead of individual object files. This causes only those object files to be linked that are actually needed, as opposed to linking object files directly, which causes them to always be included in the executable file. However, the library file must be specified after all other object files in the command line linker because it calls from the file object specified after the library file to will not be judged by the linker. Calls from inline functions to other inline functions will be handled by the linker automatically (with the option in ar rcs ensuring this). An alternative solution is to use link time optimization instead of a library. gcc provides a flag -Wl,--gc-section on the omit section in which all functions are unused. This will be the case for object files containing the code of one unused external inline function. However, it also removes all other unused sections from all other object files, not just those related to unused external inline functions. (It may be desirable to link functions to an executable file to be called by the programmer from the debugger, rather than the program itself, for example, to examine the internal state of the program.) With this approach, you can also use a single .c file with all external inline functions instead of a single .c file per function. Then the file must be compiled with the -fdata-section-ffunction-section. However, the GCC manual page warns about this, saying: Use these options only when there are significant benefits from it. Some recommend a completely different approach, which is to define functions as static nested instead of nested in header files[2]. Then no unreachable code is generated. However, this approach has a disadvantage otherwise: Duplicate code is generated if the function could not be inclineted in more than one translation unit. The issued function code cannot be shared between translation units because it must have different addresses. This is another drawback; downloading the address of such a function, defined as static nested in the header file, will yield different values in different translation units. Therefore, static nested functions should only be used if they are used in only one translation drive, which means that they should only go to the appropriate .c file, not to the header file. gnu89 gnu89 semantics inline and external inline are essentially the exact opposite of those listed in C99[4], except that gnu89 allows redefining external inline functions as unqualified functions, whereas the C99 inline is not[5]. Thus, gnu89 external inline without redefining it as C99 inline, and gnu89 inline is like c99 external inline; in other words, in gnu89, a function is always defined inline, and a function defined by an external inline never emits an externally visible function. This is because it matches variables for which storage will never be reserved if it is defined as external and always if it is defined without. The rationale for the C99, on the contrary, is that it would be surprising if the use of inline would have a side-effect of always emitting a non-inlined version of the feature–that's contrary to what its name suggests. Notes for C99 on the need to provide exactly one externally visible instance of a function inlined function and about the resulting problem with unreachable code use mutatis mutandis on gnu89 as well. gcc up to and including version 4.2 used gnu89 inline semantics, although -std=c99 was explicitly specified. [6] With version 5[5] gcc switched from gnu89 to dialect gnu11, which by default effectively allows semantics C99 inline. To use gnu89 semantics instead, they must be enabled explicitly, either with -std=gnu89 or, just affect inlining, -fgnu89-inline, or by adding the gnu_inline attribute to all nested declarations. To ensure C99 semantics, either -std=c99, -std=c11, -std=gnu99, or -std=gnu11 (without -fgnu89-inline) can be used. [3] C++ V C++, a function defined by inline if necessary, emits a function shared between translation units, usually by inserting it into a common part of the file of the object for which it is needed. The function must have the same definition everywhere, always with an inline qualifier. In C++, the external inline is the same as the inline. The rationale for the C++ approach is that it is the most appropriate method for the programmer, as no specific measures must be taken to remove the unreachable code and, as with normal functions, it is not relevant whether it is externally specified or not. The inline qualifier is automatically added to a function defined as part of a class definition. armcc armcc in C90 mode provides external inline and inline semantics that are the same as in C++: Such definitions will issue a feature shared between translation units if necessary. In C99 mode, the external inline always issues a function, but as in C++, it will be shared between translation units. This means that the same function can be defined externally inline in different translation units[7]. This corresponds to the traditional behavior of Unix C compilers[8] for multiple non-fax definitions of non-initialized global variables. Restrictions If they look at the address of a nested function, the code for not installing a copy of this feature will in any case be released. In C99, an inline or external inline function may 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 the call was made. Only static nested definitions can refer to internal link identifiers without restrictions; these will be different objects in each translation unit. In C++, both const and non-const static locals are allowed and apply to the same object in all translation units. gcc cannot nested functions if[3] are variadic, use alloca use calculated goto use nonlocal goto use nested function to use setjmp to use __builtin_return, or use __builtin_apply_args Based on Microsoft specifications on MSDN, MS Visual C++ cannot (__builtin_longjmp with the __forceinline) if the or its caller is compiled with /Ob0 (the default option for debugging builds). The function and the caller use different types of exception handling (C++ exception handling in one structured exception handling in another). The function has a list of variable arguments. The function uses nested reports if compiled with /Og, /Ox, /O1, or /O2. The function is recourse and is not accompanied #pragma inline_recursion (he). With pragmatus, recursive functions are inlined to a default depth of 16 calls. To reduce the depth of inlining, use inline_depth pragma. The function is virtual and is called practically. Direct calls to virtual functions can be inlined. The program downloads the function address and the call is made using the function pointer. Direct calls to features that had their address taken may have been inlined. The function is also marked with a __declspec modifier. Problems See also: Inline Expansion § Effect on performance In addition to problems with inline expansion in general, inline functions such as language function may not be as valuable as it seems for several reasons: Often, the compiler is in a better position than a person to decide whether a particular function should be inlined. Sometimes the compiler may not be able to nest as many features as the programmer suggests. An important point to note is that the code (inline function) gets exposed to your client (call function). As functions evolve, they may become suitable for inlining where they have not been before, or are no longer suitable for inlining where they were before. While inlining or un-inlining functions is easier than converting to and from macros, it still requires additional maintenance, which usually brings relatively little benefit. Inline features used in propagation in native C-based compilation systems can increase compilation time because the middle representation of their organs is copied to each call page. The inline specification in C99 requires exactly one external function definition if it is used somewhere. If the programmer does not make such a definition, it can easily lead to linker errors. This can happen when optimization is turned off, which usually prevents inlining. Adding definitions, on the other hand, can cause unreachable code if the programmer is not carefully avoided, by being in the library for linking, by optimizing the link time, or by static inline. In C++, you need to define an inline function in each module (translation unit) that uses it, while an ordinary function must be defined in only one module. Otherwise, it would not be possible to assemble one module independently of all other modules. Depending on the compiler, this may cause each object file concerned to contain a copy of the function code for each module with some use that could not be inclineed. built-in software, oftentimes some features must be placed in certain sections of the code using special compiler instructions such as pragma statement. Sometimes, a function in one memory segment may need to call a function in another memory segment, and if there is an inlination called a function, the code called a function may end up in a segment where it should not be. For example, high-performance memory segments can be very limited in code space, and if a feature belonging to that space calls another large function that is not designed to be in a high-performance section, and the function is inappropriately identified, it may cause the high-performance memory segment to be spent on the code space. For this reason, it is sometimes necessary to ensure that functions do not become inline. Quotation marks Declaration function [. .] with nested specifier declares the inline function. The inline specifier indicates to the implementation that inline replacement of the function of the body at the point of call should be preferred over the usual mechanism of calling the function. Implementation is not necessary to perform this inline substitution at the point of call; even if this inline replacement is omitted, the other rules for inline functions as defined in point 7.1.2 are still complied with. – ISO/IEC 14882:2011, current C++ standard, section 7.1.2 Function declared with inline function specifier is nested function. [. . .] Creating a nested feature indicates that calls to the feature are as fast as possible. The extent to which such proposals are effective is defined by implementation (footnote: For example, implementation may never perform inline substitution or may only perform inline call substitution within the nested statement range.) [. . .] A nested definition does not provide an external function definition and does not prohibit an external definition in another translation unit. A nested definition provides an alternative to an external definition that a translator can use to perform any call for a function in the same translation unit. It is not known whether the call to a function uses a nested definition or an external definition. – ISO 9899:1999(E), Standard C99, Section 6.7.4 See also Macro (Informatics) References ^ and b Meyers, Randy (1 July 2002). New C: Inline features. Quote magazine requires |journal = (help) ^ b ^ and b c d ^ ^ and b ^ ^ ^ gcc manual page, description -fno-common JANA, DEBASISH (January 1, 2005). C++ A PROGRAMOVANIE PARADIGMA. PhI Learning Pvt. Ltd. ISBN 978-81-203-2871-6.CS1 maint: ref=harv (link) Sengupta, Probal (1. augusta 2004). Objektovo orientované programovanie: Základy a aplikácie. PhI Learning Pvt. Ltd. ISBN 978-81-203-1258-6.CS1 maint: ref=harv (link) Svenk, Goran (2003). Objektovo orientované programovanie: Použitie C++ pre inžinierstvo a technológie. Cengage učenia. ISBN 0-7668-3894-3.CS1 maint: ref =harv (odkaz) Balagurusamy (2013). Objektovo orientované programovanie s C++. Tata McGraw-Hill školstva. ISBN 978-1-259-02993-6.CS1 maint: ref =harv (link) Kirch-Prinz, Ulla; Prinz, Peter (2002). Kompletný sprievodca programovania v C + +. Jones &amp; Bartlett Learning. ISBN 978-0-7637-1817-6.CS1 maint: ref = harv (odkaz) Conger, David (2006). Vytváranie hier v službe C++: Podrobný sprievodca. Noví jazdci. ISBN 978-0-7357-1434-2.CS1 maint: ref =harv (link) Skinner, M. T. (1992). Pokročilá kniha C++. Silicon Press. ISBN 978-0-929306-10-0.CS1 maint: ref = harv (odkaz) Láska (1. septembra 2005). Linux Kernel Development. Pearson školstva. ISBN 978-81-7758-910-8.CS1 maint: ref =harv (link) DEHURI, SATCHIDANANDA; JAGADEV, ALOK KUMAR; RATH, AMIYA KUMAR (8. OBJEKTOVO ORIENTOVANÉ PROGRAMOVANIE POMOCOU C++. PHI Learning Pvt. Ltd. ISBN 978-81-203-3085-6.CS1 maint: ref=harv (link) Externé odkazy Inline funkcie s GNU Compiler Collection (GCC) Získané z

roadmaster_granite_peak_24_walmart.pdf
marathi_hasya_kavita.pdf
mesupiwupisawita.pdf
the_black_cat_summary_edgar_allan_poe.pdf
add non steam game catastrophic failure
70 functions of the holy spirit pdf
ateşten gömlek pdf özet
agricultural machinery design pdf
download nkjv bible pdf file
miopia astigmatismo hipermetropia pdf
3d printing concrete pdf
english speaking course free download pdf in telugu
anvil basingstoke seating plan pdf
nutritional biochemistry pdf download
brightly beams our father' s mercy pdf
anestesia topica dental pdf
sports heads soccer unblocked games
sixth and seventh books of moses
normal_5f95fb17f383e.pdf
normal_5f8fa1ca20bd0.pdf
normal_5f8e297e9b902.pdf
normal_5f87253044786.pdf