A Few Comments on Inlining

My memory about the recent standardization of the **`inline`** keyword was in fact correct: the original Kernighan and Ritchie book (1978) contains no mention of **`inline`** in the index, although it does give definitions for inlined macros such as **`getc`** in another context.

The **`inline`** keyword was officially introduced in the ISO/IEC C99 standard, although it had been available in a number of compilers (with varying effects) before standardization. *The compiler is not required to honor the request for inlining.*

With **`gcc`**, use of the **`inline`** keyword is normally ignored unless optimization is requested. A **`gcc`** attribute (an example is shown later) can be used to force inlining even with unoptimized code, but the absence of a stack frame with an inlined function can make debugging difficult. For example, when a program is in an inlined function, the current stack frame printed by **`gdb`** corresponds to the inlined function, but the next stack frame corresponds to the caller of first non-inlined function up the call stack. In other words, in the simplest case (non-nested inlining), the call stack/backtrace/**`where`** command skips the function that calls the inlined function and shows the caller of the caller as the next frame.

To force inlining without turning on other optimizations, you can pass the **`-finline-functions`** option to **`gcc`**. There are actually four or five inlining-related flags and a variable (**`-finline-limit=`***N*) that limits the maximum 'length' (see the **`gcc`** info pages) of functions that can be inlined.

Inlined functions are most useful when the definition (the code) is placed in a header file. In C, a function defined in a header file poses the same risks as a variable defined in a header file: each compilation unit generates a separate copy. If the **`static`** keyword is not used with an inlined function, the compiler must generate a copy of the function for use from outside of the compilation unit. These copies are like normal functions, and two compilation units that include such a copy (of the same function) cannot be linked together. If an inlined function

is marked **`static`**, but inlining is not done, the functions are still produced, but the symbols are marked as "weak," which means that the compiler is allowed to silently discard all but one copy. The linker does not check that the code generated in these copies is identical, so *trying to tailor inline functions on a per-compilation unit basis can be risky.*

With C++, member functions defined directly in a class are implicitly treated as though marked with the **`inline`** keyword, and are thus inlined when compiled at any level of optimization (above 0); this behavior can be turned off with **`-fno-default-inline`**.

The following C function and C++ member function are inlined at any optimization level (that is, **`-O1`** or higher; **`-O0`** means no optimizations).

```
static inline int
func (int a)
{
    printf ("%d\n", a);
}
class spot {
    private: int x;
    public: void show () { printf ("%d\n", x); }
};
```

The following C function is always inlined (the attribute seems to override even command-line options such as **`-fno-inline`**):

```
static inline __attribute__ ((always_inline)) int
func (int a)
{
    printf ("%d\n", a);
}
```