

C

Using the Optional Inliner

This appendix describes an optional compiler pass called an *inliner* of routines. It contains the following sections:

§C.1: *Overview*

§C.2: *Invoking the Inliner*

§C.3: *Inliner Option Reference*

§C.4: *How the Inliner Constructs Unique Names*

§C.5: *Pragmas for Inlining*

C.1 Overview

*See the **High C/C++ Language Reference** for information on iterators, a High C/C++ extension.*

The inliner replaces calls to functions and iterators (hereafter called *routines*) with the logic contained within the routines, thus avoiding the overhead of function calls at the expense of potentially larger code size.

You can specify particular routines to be inlined, and also allow the inliner to automatically select routines that satisfy adjustable criteria, such as the number of calls and the size of the routine.

Inlining saves overhead

The inliner back-substitutes routine bodies where the routines are called, and replaces parameters with local variables or constants where appropriate. This saves at least the routine prolog, epilog, and parameter-passing overhead, and exposes opportunities for additional optimizations — such as copy propagation and constant folding — especially when the arguments themselves are constants.

Some inlined routines must also be defined

Factors such as exportability, addressability, and recursiveness play roles in determining whether an inlined routine must also be defined. If the routine can be completely inlined, its definition is not emitted. See §C.2.1: *How the Compiler Selects Routines for Inlining*. Operationally, the inliner sits between the compiler's front end and the common optimizing back end.

Note: The inliner described here is *not* necessary for compiling C++ functions, even those with in-line function declarations. Those functions are inlined by the C++ compiler's built-in inliner, which offers almost the same functionality as the optional inliner described here.

Use this optional inliner for C++ programs only if you want more inlining than is provided by the C++ compiler. See §8.2: *Using the Built-In Inliner Versus the Stand-Alone Inliner*, and §6.5.17: *Inlining ANSI Standard C Functions*.

C.2 Invoking the Inliner

You specify inliner options on the driver command line to invoke the inliner during compilation. The compiler invokes the inliner implicitly at optimization levels 4 and higher.

Use **-Hix** options To activate the inliner to do the following:

to activate the
inliner

- achieve automatic inlining
- prevent selected categories of routines from being inlined
- customize the inlining process

specify an appropriate optimization level with option **-O** (as described in §6.5.18: *Inlining Functions*) or use one or more of the **-Hix** options (described in §C.3: *Inliner Option Reference*).

Use type qualifier **_Inline** to inline individual functions If you specify command-line option **-Hi** without a suffix, you must use type qualifier **_Inline** in your source code to request inlining for individual functions. For example:

```
_Inline void func() { ... }
```

causes `func()` to be inlined when you compile with **-Hi** on the command line. Also, you can explicitly specify **_Inline** for a function, to override any **-Hix** command-line option that might otherwise exclude that function from inlining (see §C.3: *Inliner Option Reference*).

C.2.1 How the Compiler Selects Routines for Inlining

<i>Type qualifier</i>	The options that determine inlinability (described in §C.3: <i>Inliner Option Reference</i>) are overridden if you explicitly designate a routine as inlined with the <code>_Inline</code> type qualifier.
<i><code>_Inline</code> overrides inlining options</i>	
	The <code>-Hit</code> and <code>-Hic</code> options work together. If both are specified and their specified arguments are not 0 (zero), both conditions must be met. If only one is specified, and the specified condition is met, the routine is inlined.
<i>Pragmas control inlining of individual functions</i>	You control the inlining of specific functions with pragmas <code>On_inline</code> and <code>Off_inline</code> . When activated, these pragmas are pushed onto a stack for each function specified. Use pragma <code>Pop_inline</code> to pop the top-most stack entry for the specified function and restore the previous state for that function. See §5.2: <i>Pragma Reference</i> for more information.

C.2.1.1 Rules for Inlining Routines

- Routines chosen for inlining that are exported, recursive, or have their address taken are inlined but also defined.
- Routines chosen for inlining that are not exported, not recursive, and do not have their address taken, are inlined but the definition is discarded.
- Recursive routines are inlined only once and then a call is generated.
- Routines containing nested routines can be inlined only if all nested routines are inlined.
- Routines containing calls to the intrinsic function `_Alloca()` or to an external routine named `alloca()` are not inlined.
- Routines that use variable arguments are not inlined.

C.2.2 Debugging Inlined Routines

<i>Option <code>-g</code> suppresses inlining</i>	If you specify <code>-g</code> on the command line, the inliner is suppressed. Reasonable debugging is not possible in the presence of inlined routines, especially when you use multi-module inlining.
---	---

C.3 Inline Option Reference

Note: Many of the options listed in this section are implicitly specified at optimization level 4, but can be explicitly specified at any optimization level to alter the compiler's default behavior.

-Hi — Invoke the inliner during compilation

Use this option when you want to inline only those routines you designate individually with type qualifier **_Inline**, and when you are not specifying any other **-HiX** option on the command line.

-Hia — Do not inline functions that have their address taken

Routines that have their address taken might be inlinable, but still must be defined so that they will have an address.

-Hib=n — Specify temporary-file buffer size

The buffer size for the temporary files is set at 65,536. You can override this by specifying *n* as something different; for example, smaller to save buffer space. The default buffer size is large because the inliner does a lot of seeking/reading through the temporary files.

-Hic=n — Inline functions called fewer than *n* times

Use -Hic to inline only infrequently called routines When you specify **-Hic** and *n* is not 0 (zero), routines that are directly called fewer than *n* times are automatically inlined. If you specify **-Hic** and **-Hit** (with non-zero values), they work together — that is, both conditions must be met before a routine is automatically inlined.

The compiler default value for *n* is 4. The default value is used only when the optimization level is 4 or higher, and you do not specify a value for *n* on the command line.

-HiC=s, n — Inline functions smaller than *s* and called fewer than *n* times

Option **-HiC** specifies that a function with fewer than *s* nodes should be inlined only if it is called fewer than *n* times. You can use **-HiC** more than once to specify functions of different sizes and number of calls. For example:

```
hc -HiC=10,5 -HiC=20,4 -HiC=25,2 myprog.c
```

This example tells the compiler to inline the following:

- any function with fewer than 10 nodes if it is called fewer than 5 times
- any function with fewer than 20 nodes if it is called fewer than 4 times
- any function with fewer than 25 nodes if it is called fewer than 2 times

Note: Do not confuse option **-HiC** with option **-HiC**, which specifies inlining only infrequently called routines.

-Hih — Show which functions have been inlined and/or defined

Option **-Hih** displays the name and status of each function as the inliner encounters it. The status information tells whether the function has been inlined, inlined and defined, or not inlined. If a routine is not inlined, the status information includes the reason.

This option also prints the following information for each routine:

- the size of the routine, in tree nodes
- an approximation of the stack-frame size required by the routine
- the number of calls made to the routine

Note: There is approximately one tree node for each operator, operand, or other language construct such as a statement, loop, declaration, or function.

-Hir — Do not inline recursive functions

If a directly or indirectly recursive function meets the criteria for inlining, it is inlined only at the first level and then a definition is generated so that it can be called at subsequent levels.

You might want to avoid inlining recursive functions. The **-Hir** option suppresses inlining of recursive functions even at the first invocation level.

-His=*n* — Inline functions with stack size less than *n* bytes

When you specify **-His** (*n* is not zero), only routines with a stack size of less than *n* bytes are automatically inlined.

-Hit=*n* — Inline functions with fewer than *n* tree nodes

Use -Hit to inline only simple routines When you specify **-Hit** and *n* is not 0 (zero), routines with fewer than *n* expression-tree nodes are automatically inlined. There is one tree node,

roughly, for each operator, operand, or other language construct, such as a statement, loop, declaration, or routine.

We provide a default value for n in the `.cnf` file. The default value is used only when the optimization level is greater than or equal to 4 and you do not specify a value for n on the command line.

-Hiw — Invoke inliner in multi-module (wide inlining) mode

Wide inlining With the **-Hiw** option, the inliner operates on all files being compiled as a collection rather than on each file individually. This is known as *multi-module inlining* or wide inlining.

Multi-module inlining permits inlining a function from one module into another module. This can improve performance when you use separate files for data hiding, or when more than one programmer is working on a project.

Benchmark performance can also be improved in cases where a benchmark is split across multiple files. For example, you can realize a significant performance improvement in the Dhrystone 2.1 benchmark with the following options:

```
hc dhry1.c dhry2.c -Hiw -Hit=250 -Hwide -o
```

Use -Hwide also for wide compilation To achieve multi-module inlining, you must also specify command-line option **-Hwide**, for “wide” compilation. See Chapter 3: *Using Compiler Options* for more information about this option.

-Hix — Do not inline exported functions

Exported routines might be inlinable, but they still must be defined in order to be exported.

C.4 How the Inliner Constructs Unique Names

For multi-module inlining, the inliner invents public names to communicate private data or code from one module to another. You might encounter these names when you use a debugger (although debugging inlined code is not recommended). The names are constructed as follows:

When function f in one module $m1$ that uses a data item D is inlined into separate module $m2$, data item D must be available to $m2$.

If `D` is private to `m1`, the inliner must make it public so that `m2` can access it.

Here is an example of the inliner constructing unique names:

File m1.c:

```
static int i = 7;
sub(int j) { i += j; }
```

File m2.c:

```
main () {
    sub(1);
}
```

In this example, `i` must be available to the body of `main()` in `m2` when `sub()` is inlined. The inliner constructs a unique name for `i` and makes it public; that name is then used within both `m1` and `m2`.

The inliner constructs a unique name that can be easily traced back to the defining module. The constructed name consists of four components, in this order:

- a double underscore “__”
- the original variable/function name
- an underscore followed by the module defining the variable/function
- an underscore followed by a checksum value

For example, the unique constructed name for the private variable `i` in module `m1` of the preceding example would be `__i_m1_Vbd9755ee3`.

In the case of private compiler-generated data, `in1` is used in place of the variable name.

C.5 Pragmas for Inlining

`off_inline(function_name[, function_name ...])` — Turn off inlining for specific functions

When this pragma is active, it prevents calls to `function_name` from being inlined when they otherwise would be, even if the function is declared as an

inline function, or, in the case of C++ class member functions, is inlined in the class definition.

The `Off_inline` state is pushed onto a stack for each function specified.

On_inline(*function_name*[, *function_name* . . .]) — **Turn on inlining for specific functions**

When this pragma is active, it causes calls to *function_name* to be inlined even if a function does not otherwise meet the criteria for inlining (for example, if it is not declared as an **inline** function, or if it does not fit the size or call-count requirements for inlining of C functions).

C++ only Pragma `On_inline` has no effect on inlinable C++ member functions. If C++ functions are to be inlined at all, they must be declared as **inline** functions or inlined directly in the class definition. Such functions are then inlined automatically, unless their inline status has been turned `Off` with pragma `Off_inline`.

With C-like functions in C++ code, pragma `On_inline` operates just as it does on regular C functions.

Note: Even if a function appears in the `On_inline` argument list, it cannot be inlined if it contains calls to `alloca()` or to `setjmp()`, or if it is a `varargs` function.

Pragma `On_inline` does, however, override inliner options that suppress inlining for recursive, exported, and nested functions, and functions that have their address taken.

The `On_inline` state is pushed onto a stack for each function specified.

Pop_inline(*function_name*[, *function_name* . . .]) — **Pop topmost stack entry for each specified function**

This pragma pops the topmost stack entry for each function specified, and restores the previous `On_inline` or `Off_inline` state.