6.3.1.3 Optimization Levels

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the subprogram and get exactly the results you would expect from the source code.

Turning on optimization makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

If you use multiple -O options, with or without level numbers, the last such option is the one that is effective.

The default is optimization off. This results in the fastest compile times, but GNAT makes absolutely no attempt to optimize, and the generated programs are considerably larger and slower than when optimization is enabled. You can use the `-O` switch (the permitted forms are `-O0`, `-O1` `-O2`, `-O3`, and `-Os`) to `gcc` to control the optimization level:

*

`-O0`

No optimization (the default); generates unoptimized code but has the fastest compilation time.

Note that many other compilers do substantial optimization even if 'no optimization' is specified. With gcc, it is very unusual to use `-O0` for production if execution time is of any concern, since `-O0` means (almost) no optimization. This difference between gcc and other compilers should be kept in mind when doing performance comparisons.

*

`-O1`

Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly.

*

`-O2`

Full optimization; generates highly optimized code and has the slowest compilation time.

*

`-O3`

Full optimization as in `-O2`; also uses more aggressive automatic inlining of subprograms within a unit ([Inlining of Subprograms], page 206) and attempts to vectorize loops.

*

`-Os`

Optimize space usage (code and data) of resulting program.

Higher optimization levels perform more global transformations on the program and apply more expensive analysis algorithms in order to generate faster and more compact code. The

price in compilation time, and the resulting improvement in execution time, both depend on the particular application and the hardware environment. You should experiment to find the best level for your application.

Since the precise set of optimizations done at each level will vary from release to release (and sometime from target to target), it is best to think of the optimization settings in general terms. See the 'Options That Control Optimization' section in *Using the GNU Compiler Collection (GCC)* for details about the `-O` settings and a number of `-f` options that individually enable or disable specific optimizations.

Unlike some other compilation systems, `gcc` has been tested extensively at all optimization levels. There are some bugs which appear only with optimization turned on, but there have also been bugs which show up only in 'unoptimized' code. Selecting a lower level of optimization does not improve the reliability of the code generator, which in practice is highly reliable at all optimization levels.

Note regarding the use of `-O3`: The use of this optimization level ought not to be automatically preferred over that of level `-O2`, since it often results in larger executables which may run more slowly. See further discussion of this point in [Inlining of Subprograms], page 206.

6.3.1.4 Debugging Optimized Code

Although it is possible to do a reasonable amount of debugging at nonzero optimization levels, the higher the level the more likely that source-level constructs will have been eliminated by optimization. For example, if a loop is strength-reduced, the loop control variable may be completely eliminated and thus cannot be displayed in the debugger. This can only happen at `-O2` or `-O3`. Explicit temporary variables that you code might be eliminated at level `-O1` or higher.

The use of the `-g` switch, which is needed for source-level debugging, affects the size of the program executable on disk, and indeed the debugging information can be quite large. However, it has no effect on the generated code (and thus does not degrade performance)

Since the compiler generates debugging tables for a compilation unit before it performs optimizations, the optimizing transformations may invalidate some of the debugging data. You therefore need to anticipate certain anomalous situations that may arise while debugging optimized code. These are the most common cases:

* 'The 'hopping Program Counter':' Repeated `step` or `next` commands show the PC bouncing back and forth in the code. This may result from any of the following optimizations:
  - 'Common subexpression elimination:' using a single instance of code for a quantity that the source computes several times. As a result you may not be able to stop on what looks like a statement.
  - 'Invariant code motion:' moving an expression that does not change within a loop, to the beginning of the loop.
  - 'Instruction scheduling:' moving instructions so as to overlap loads and stores (typically) with other code, or in general to move computations of values closer to their uses. Often this causes you to pass an assignment statement without the assignment happening and then later bounce back to the statement when the value is actually needed. Placing a breakpoint on a line of code and then stepping over it may, therefore, not always cause all the expected side-effects.