

Using Compiler Toggles

This chapter documents High C/C++ compiler toggles and describes how to use them. It contains the following sections:

§4.1: *Specifying Toggles*

§4.2: *Toggle Reference*

4.1 Specifying Toggles

Toggles are switches that turn compiler controls `On` or `Off`. You can specify toggles on the command-line, in which case they affect the entire program being compiled. Alternatively, you can use them in your source code to affect only certain portions of your code.

On the command line To set or reset a toggle on the command line, you use option `-Hon` or `-Hoff` (see Chapter 3: *Using Compiler Options*). For example:

```
hc -Hon=Toggle_name ...
hc -Hoff=Toggle_name ...
```

In source code or in a profile To set a toggle in source code or in a profile, you use a compiler pragma (see Chapter 5: *Using Compiler Pragas*). For example:

```
#pragma On(Toggle_name)
#pragma Off(Toggle_name)
#pragma Pop(Toggle_name)
```

Toggle settings are stacked Pragma `On` turns the toggle `On`; pragma `Off` turns it `Off`; and pragma `Pop` reinstates it to a prior value. Toggles operate in a stack-like fashion, where each `On` or `Off` is a “push” of the previous state, and a `Pop` “pops” the stack.

```
#pragma On (List) /* Turns On source listing */
#pragma Off(Cleanup_spills)
/*      Register spill code not cleaned up */
#pragma Off(List) /* Turns Off source listing */
#pragma On (List) /* Turns On source listing */
#pragma Pop(List) /* Back to Off for listing */
#pragma Pop(List) /* Back to On for listing */
```

Note: Toggle names are not case-sensitive.

4.2 Toggle Reference

Auto_class_member_instantiation — Enable automatic instantiation of class templates

Default: **on**

C++ only In a C++ program, a class template must always be instantiated when it is used. By default, the compiler generates instances of class members when the class is used in a source module, and when the definitions of those members are likewise presented to the compiler. To disable automatic instantiation, turn **Off** toggle **Auto_class_member_instantiation**.

See Appendix E: *Manual Template Instantiation*.

Auto_func_instantiation — Enable automatic instantiation of function templates

Default: **on**

C++ only When you use the same template in separate source modules in a C++ program, the linker returns an error complaining of a duplicate definition. To prevent duplicate definitions, disable automatic template instantiation by turning **Off** toggle **Auto_func_instantiation**.

See §8.8.1: *The Single-Copy Problem*.

Autodebug — Place named SDI for a named **struct** in the **struct**'s debug repository

Default: **off**

When toggle **Autodebug** is **On** and you specify option **-g**, the compiler places named symbolic debug information (SDI) in the debug repository of any named C or C++ **struct** (including unions and classes) declared within the toggle's scope.

The debug repository of a **struct** is the compilation unit in which the **struct**'s first non-pure-virtual, non-inline function is defined.

See §13.3: *Minimizing Symbolic Debug Information* for more information.

Note: Toggle **Autodebug** applies only if toggle **Dwarf** is **On**.

Back_substitute_epilog — Replace unconditional jump to a function epilog with epilog sequence

Default: **off** unless optimization level ≥ 2

When toggle `Back_substitute_epilog` is On, the compiler replaces an unconditional jump to the epilog of a function with the epilog sequence.

See §6.5.2: *Back-Substitution of Epilog Code* for more information.

Back_substitute_nodes — Reduce jumping due to conditional blocks within a loop

Default: **off** unless optimization level ≥ 4

When toggle `Back_substitute_nodes` is On, the compiler tries to reduce the number of jumps due to conditional blocks inside a loop construct.

Note: Toggle `Back_substitute_nodes` is turned Off when you compile with command-line option **-g**.

Behaved — Specify a “well-behaved” program

Default: **off** unless optimization level ≥ 4

Toggle `Behaved`, when On, specifies that the program is “well-behaved,” and the optimizer can be less conservative in generating code to reference pointer-based objects.

A *well-behaved* program is one that follows these rules:

- The address of a union member is never assigned to a pointer.
- A value of a pointer type is never cast to an incompatible pointer type.

With these assumptions, the compiler might be able to generate substantially better code in referencing pointer-based variables. The compiler issues an appropriate warning if either of these assumptions is violated in such a way as to affect assumptions made by the optimizer.

You must decide whether the warnings can be safely ignored or whether the program should be compiled at a lower optimization level.

Caution: The compiler might not catch all instances of misbehaved code. For example, a pointer-to-**char** might be passed to an

undeclared (unprototyped) external function expecting a pointer-to-**int**.

Therefore, it is possible for a program to compile at optimization level 4 without warnings (and run incorrectly), but run correctly when compiled at a lower optimization level.

Borland — Relax some ARM C++ standards to conform to the Borland class library

Default: **off**

C++ only In C++ compilations, toggle Borland relaxes some compile-time error checking, and ensures that the type of **sizeof** is the same as **size_t**.

The following operations do not conform to the **Annotated C++ Reference Manual**, but they are relaxed when this toggle is On because they appear in Borland class-library source:

- ANSI Standard C requires the type of **sizeof** to be the same as **size_t**. However, **size_t** is **unsigned**, and code such as this:

```
if (amount_left - sizeof(thing_to_allocate)) > 0
```

does not work if the type of **sizeof** is **size_t**. In particular, for High C/C++, **sizeof** is **signed**.
- In a conditional expression such as **X?Y:Z** or **X?Z:Y**, the **Y** can be a **const** structure type **T**, and **Z** a non-**const** **T**; the result type is **T**.

Call_trace — Generate information about a function being called

Default: **off**

When toggle **Call_trace** is On, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is called.

See §13.2: *Tracing Function Calls* for more information.

Char_default_unsigned — Make char default type unsigned char

Default: **off** for Solaris and Workplace UNIX targets; **On** for others

When toggle **Char_default_unsigned** is On, type **char** is **unsigned** by default.

Type **char** has one of two representations: **signed** or **unsigned**. When the representation of **char** is not specified explicitly using the keyword

signed or **unsigned**, a default representation is used. What this default is depends on whether toggle `Char_default_unsigned` is `On` or `Off`:

- If the toggle defaults to `On`, the default for **char** is **unsigned**.
- If the toggle defaults to `Off`, the default for **char** is **signed**.

The default setting of toggle `Char_default_unsigned` is ordinarily **unsigned**, but can be altered from the configuration (`.cnf`) file.

Use this test program to determine the default for your target platform:

```
main() {
    printf("%d\n", (char) - 1);
}
```

If the test program prints `-1`, the default for the toggle is **signed**. If the program prints `255`, the default for the toggle is **unsigned**.

Regardless of the default representation, type **char** is still not identical to either type **unsigned char** or type **signed char**. The compiler issues a warning, for example, if the default representation of **char** is **unsigned** and you try to assign a variable of type **unsigned char*** to type **char***.

See also toggle `Char_is_rep`.

See §9.1.1: *Characters* for more information about type **char**.

Char_is_rep — **Make the default representation of char identical to either signed char or unsigned char**

Default: `off`

Type **char** has one of two representations: **signed** or **unsigned**. For example, when **char** is **unsigned**, `0x80` represents `128`, but when **char** is **signed**, `0x80` represents `-128`.

According to the ANSI C Standard, **char**, **signed char**, and **unsigned char** are three distinct types. Therefore type **char*** cannot be assigned to type **signed char*** or type **unsigned char***. The compiler issues an error or a warning if you attempt such an assignment.

When you do not explicitly specify the representation of **char** using the keyword **signed** or **unsigned**, the compiler uses a default representation. This default depends on the compiler implementation. Even though the default representation of **char** is **signed** or **unsigned**, however, the

compiler does not recognize it as being the same type as **signed char** or **unsigned char**.

If you turn On toggle `Char_is_rep` (Off by default), the compiler recognizes type **char** as being identical to the default representation chosen for **char** — either **unsigned char** or **signed char**. For example, if `Char_is_rep` is On and the default representation of **char** is **unsigned char**, a variable declared as type **unsigned char*** can be assigned to type **char*** without the compiler issuing a warning or error.

Note: Be sure to turn On toggle `Char_is_rep` before any you make any declarations.

Caution: Turning On toggle `Char_is_rep` might decrease program portability.

See also toggle `Char_default_unsigned`.

See §9.1.1: *Characters* for more information about type **char**.

Cleanup_spills — Clean up register spill code

Default: **On** if optimization level ≥ 3 ; **Off** if optimization level < 3

When toggle `Cleanup_spills` is On, the compiler performs another iteration of local common subexpression elimination to clean up register spill code, if possible. For additional information, see §6.5.37: *Spill-Code Clean-Up*.

Command_line_ident — Put information into object code's `.comment` section

Default: **On**

When toggle `Command_line_ident` is On, the compiler generates the assembler directive `.ident` to insert information into the object code's `.comment` section.

This inserted information includes the compiler's version number. Generating this information is not necessary, so it is safe to turn the toggle Off.

Common_can_export — Allow public variables in common segmentsDefault: **off**

When toggle `Common_can_export` is **On**, the compiler compiles code such that other modules can “see” global variables declared in the data section created with `pragma Data`.

When toggle `Common_can_export` is **On**, public variables are permitted in common sections. When it is **Off**, variable names in a common section are *not* made public.

Const_in_code — Determine how `const` static variables map in storageDefault: **off**

Toggle `Const_in_code` determines how static variables, exported or local, are to be mapped in storage when they are declared with the **const** qualifier as in this example:

```
static const char x[] = "ABCDEF" ;
```

(By definition, such variables cannot be modified at run time.)

When toggle `Const_in_code` is **Off**, **const** static variables are placed in the *literals* section. The default name for the literals section is `.rdata`, but you can change the name with the `Literals` pragma (see Chapter 5: *Using Compiler Pragmas* and §6.5.8: *Common Subexpression Elimination (Local)*).

When toggle `Const_in_code` is **On**, **const** static variables are mapped into the default code section.

Cross_jump — Allow cross-jumping (tail-merging) optimizationDefault: **On** if optimization level > 0; **Off** if optimization level = 0

When toggle `Cross_jump` is **Off**, the cross-jumping (tail-merging) optimization is suppressed. Suppressing cross jumping can make code easier to debug at the instruction level.

Note: Toggle `Cross_jump` is turned **Off** when you specify option **-g** or **-O0**.

For more information, see §6.5.14: *Cross Jumping (Tail Merging)*.

Dbx — Generate debug information in SUN dbx `stab` formatDefault: **off**

When toggle `Dbx` is On and you specify option `-g`, the compiler generates debug information in the SUN **dbx stab** format.

Define_static_members — Supply implicit definition of static class data members

Default: **On**

C++ only According to the C++ language definition, static data members of a class must be defined outside the class.

```
class s {
    static int x; /* This is NOT a definition */
};
int s::x;        /* This is the required definition */
```

Turning On toggle `Define_static_members` relaxes this restriction, and the definition is implicitly supplied by the compiler. The toggle accommodates some existing practices.

See Chapter 8: *C++-Specific Issues*.

Demote_sizes — Allow == operator to compare integer types other than int

Default: **On** for optimization levels ≥ 5 ; **Off** otherwise

When toggle `Demote_sizes` is On, operator `==` can take operands of type **unsigned short**, **signed short**, and **char** without casting them to **int**.

Note: Toggle `Demote_sizes` is turned Off when you compile with command-line option `-g`.

Caution: Compiling with toggle `Demote_sizes` turned On might cause your code to run more slowly.

Double_math_only — Perform double-precision floating-point arithmetic

Default: **off**

When toggle `Double_math_only` is On, floating-point operations are performed in double precision.

When two operands of certain arithmetic operations are both of type **float**, the ANSI C Standard permits an implementation to do one of two things:

- perform the operation using **float** arithmetic, in which case the result of the operation is of type **float**
- convert both operands to type **double** and use **double** arithmetic, in which case the result of the operation is of type **double**

When toggle `Double_math_only` is turned `Off`, the compiler uses the first option. When it is turned `On`, the second is used instead.

Note: Option **-fsingle** option initializes toggle `Double_math_only` to `Off`. **-fdouble** initializes the toggle to `On`.

Double_return — **Make non-prototype functions return type double, not type float**

Default: `off`

When toggle `Double_return` is `On`, any non-prototype function returning type **float** returns type **double** instead. This convention conforms to some Portable C Compiler implementations.

Downshift_file_names — **Convert include file names to all lowercase**

Default: `off`

When toggle `Downshift_file_names` is `On`, the file-name specification of any subsequent **#include** directive is interpreted as if it were in all lowercase.

Toggle `Downshift_file_names` is useful when moving source code from an operating system in which file-name casing is not significant to a system in which it is significant.

Dwarf — **Generate debug information in DWARF format**

Default: `On`

When toggle `Dwarf` is `On` and you specify option **-g**, the compiler generates debug information in DWARF format (for debuggers that support DWARF).

Empty_is_void — Force C++ (void) interpretation of function declarations with no argumentsDefault: **On**

C++ only In C++, a function declared `int f();` means the same as `int f(void);` — the function takes no arguments. In C this is not the case; a function declared `int f();` can take any arbitrary arguments.

When **Off**, toggle `Empty_is_void` forces the C, not C++, interpretation of such function declarations. This is useful in “wrapping” old C header files that might contain such declarations, when compiling with C++:

```
extern "C" {
    #pragma Off(Empty_is_void)
    #include "old_hdr.h"
    #pragma Pop(Empty_is_void)
}
```

Toggle `Empty_is_void` is turned **Off** in `c_wrap1.h`.

Enforce_access_control — Allow compiler-enforced access controlDefault: **On**

C++ only In C++ compilations, turn **Off** toggle `Enforce_access_control` to remove all compiler-enforced access control. This removal can be useful when, for example, you are in a debugging mode and want to print out some private members of a class. Normally, access control might deny you access to those members.

Ensure_instantiation — Force instantiation of all template typesDefault: **Off**

C++ only Toggle `Ensure_instantiation` forces instantiation of any template types used within its scope, whether or not toggle `Auto_class_member_instantiation` is turned **On**.

See also the related pragmas, `Ensure_instantiation` and `Exclude_instantiation`, described in Appendix Example E.1: *Turning Off Automatic Instantiation*.

Epilog_trace — Generate information about a function being exitedDefault: **off**

When toggle `Epilog_trace` is On, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is exited.

See §13.2: *Tracing Function Calls* for more information.

Exception_aware_class — Compile exception-aware classesDefault: **off**

C++ only Toggle `Exception_aware_class` causes all classes compiled within its scope to be exception-aware. That is, whenever such a class is constructed, its constructor increments the global subobject count by 1. When such a class is destroyed, its destructor decrements the global subobject count by 1.

Exception-aware classes permit High C++ exception-handling (as distinct from system exception-handling).

See the **High C/C++ Language Reference** and §8.4: *Using Exception Handling* for details on exception handling.

Export_vtabs — Make virtual function tables public and definedDefault: **off**

C++ only Toggle `Export_vtabs` affects C++ classes that contain only inlined and pure virtual functions. If `Export_vtabs` is On, the virtual-function tables for such classes are made public and defined, and can be referenced from any module compiled with toggle `Import_vtabs` On.

Use toggle `Export_vtabs` in conjunction with toggle `Import_vtabs` to force only one copy of virtual-function tables for these classes to be defined in a program.

See also toggle `Import_vtabs`.

Fast_virtual_bases — Speed access to members of virtual basesDefault: **off**

C++ only Turning On toggle `Fast_virtual_bases` can improve speed of access to members of virtual bases of a class when a class hierarchy contains virtual bases as bases of other virtual bases.

When a virtual-base class B is accessible only as a base of another virtual-base class, access to B requires two pointer indirections. In general, multiple pointer indirections can be required for more complicated class hierarchies.

One way to improve the speed of access is to turn On toggle `Fast_virtual_bases`. With the toggle On, the compiler guarantees that a class has enough copies of virtual-base pointers directly accessible within the class so that access to any virtual base of the class requires only a single pointer indirection.

Toggle `Fast_virtual_bases` can be turned On and Off around class definitions; turn it On just before the definition of a class for which you want fast virtual-base access.

You can modify your C++ source to use fast virtual-base access on a case-by-case basis. For example:

```
class B { int x; };
class X : virtual B { };
class D : virtual X { };
D *dp;
```

Here, `dp->x` would require two virtual-base pointer indirections; the first to access X from D, and the second to access B (and thus X) from X. You can use the toggle, as just described:

```
class B { int x; };
class X : virtual B { };
#pragma On(Fast_virtual_bases)
class D : virtual X { };
#pragma Pop(Fast_virtual_bases)
D *dp;
```

But you can also modify the class definition of D to specify B as a direct virtual base of D. Direct virtual bases are always accessible via one virtual-base pointer indirection.

```
class B { int x; };
class X : virtual B { };
class D : virtual X, virtual B { };
D *dp;
```

In each of the last two examples, `dp->x` requires only one virtual-base pointer indirection.

Caution: If you leave toggle `Fast_virtual_bases` turned On indiscriminately, your program requires more space.

Forceddebug — Place named SDI for any in-scope structure in current compilation unit

Default: `off`

Toggle `Forceddebug` causes the current compilation unit to become the debug repository for any named structure with no natural debug repository.

When this toggle and toggle `Nodebug` are On and you specify option `-g`, the compiler places named symbolic debug information (SDI) in the current compilation unit, for any named C or C++ structure (including unions and classes) declared within the toggle’s scope. The current compilation unit becomes the debug repository for these structures.

See §13.3: *Minimizing Symbolic Debug Information* for more information.

Note: Toggle `Forceddebug` applies only if toggle `Dwarf` is On.

FP_varargs — Always save floating-point argument registers

Default: `off`

If you know your application uses floating-point variable arguments (“varargs”), you can turn On toggle `FP_varargs` to improve floating-point performance. When this toggle is On, CR1 (the condition register bit 6) is neither set nor unset, and floating-point arguments in a variable-argument function are always saved.

Note: Turning On toggle `FP_varargs` implies turning Off toggle `Non_FP_varargs`. However:

- Turning Off toggle `FP_varargs` does not imply turning On toggle `Non_FP_varargs`.
 - Turning toggle `Non_FP_varargs` On or Off has no effect on the status of toggle `FP_varargs`.
-

Caution: The standard MetaWare libraries are compiled with toggle `FP_varargs` turned Off. If you turn this toggle On for code calling library functions that use variable arguments, you should supply your own libraries.

Global_CSE — Eliminate global common subexpressions

Default: **off** unless optimization level ≥ 2

When toggle `Global_CSE` is On, the compiler performs global common subexpression elimination.

See Chapter 6.5.7: *Common Subexpression Elimination (Global)* for more information.

Note: For very large functions, the compiler might suppress global common subexpression elimination even if toggle `Global_CSE` is On.

Caution: Do not turn On toggle `Preload_args_from_memory` when toggle `Global_CSE` is On. If you turn On both toggles, the compiler still generates correct code but might tie up registers unnecessarily.

Globals_volatile — Do not move expressions with global or external variables out of loops

`Globals_volatile` is obsolete. It has been replaced by toggle `Keep_global_refs_in_loops`.

High_level_scheduling — Perform instruction scheduling before register allocation

Default: **off** unless optimization level ≥ 2

When toggle `High_level_scheduling` is On, the compiler performs instruction scheduling before it performs register allocation.

Import_vtabs — Assume virtual function tables to be external for classes with inlined, pure virtual functions

Default: **off**

C++ only In C++ compilations, toggle `Import_vtabs` applies to classes that contain inlined, pure virtual functions. If `Import_vtabs` is On, the virtual-function tables for such classes are assumed to be defined in some other module. Use toggle `Export_vtabs` to force this definition in the defining module.

Use toggle `Import_vtabs` in conjunction with toggle `Export_vtabs` to force only one copy of virtual-function tables for these classes to be defined in a program.

See §10.7.1: *Virtual-Function Tables* for more information about how the compiler produces virtual-function tables.

Induction_analysis — Readjust loop counter to eliminate compares

Default: **Off** unless optimization level ≥ 5

When toggle `Induction_analysis` is **On**, the compiler analyzes loops, and in some cases readjusts the loop counter to eliminate the compare instruction, thus speeding up your program.

See Chapter 6.5.24: *Loop Induction-Variable Elimination*.

Inline_common — Solve the single-copy problem for inline functions

Default: **On**

C++ only When toggle `Inline_common` is **On**, it solves the single-copy problem for the following types of inline functions:

- those whose bodies must be generated (for example, when the function is too big to inline, or when its address is taken)
- those whose bodies are defined by the user within the scope of the toggle

This includes inline functions within classes, and global functions declared **extern inline**.

For a discussion of the single-copy problem and its solutions, see §8.8.1: *The Single-Copy Problem*.

Int_function_warnings — Warn about missing and mistyped returns

Default: **Off**

When toggle `Int_function_warnings` is **On**, the compiler warns about missing returns or mistyped returns. Such warnings occur in the following cases:

- when a function returning an **int** has no **return expression;** statement within it
- when a function returning an **int** contains a **return** statement with no argument (that is, **return;**)

When this toggle is **Off**, the compiler suppresses these warning messages.

Note: Old C programs that do not use the reserved word **void** to indicate a function returning no result often generate many

warning messages. Leave toggle `Int_function_warnings` turned `Off` if you do not want to see the warning messages.

Keep_global_refs_in_loops — Do not move expressions with global or external variables out of loops

Default: `off`

Note: Toggle `Keep_global_refs_in_loops` was formerly called `Globals_volatile`.

When toggle `Keep_global_refs_in_loops` is `On`, the optimizer does not move expressions involving globally or externally defined variables out of loops.

Caution: Turning `On` toggle `Keep_global_refs_in_loops` does *not* have the same effect as declaring all globally or externally defined variables with the **`volatile`** type specifier.

When you declare a variable with the **`volatile`** type specifier, the optimizer does not retain that variable's value in a register across statement boundaries.

The optimizer also updates **`volatile`** variables in memory as soon as an assignment occurs, and does not delete what appear to be “dead” or redundant assignments to these variables.

Turn `On` toggle `Keep_global_refs_in_loops` if your program relies on signal functions to modify static or global variables. You can also specifically declare such variables with the **`volatile`** attribute.

List — Generate a source listing

Default: `off`

When toggle `List` is `On`, the compiler writes a listing to standard output. `List` is typically specified at the start of compilation, but it can appear in the source file to turn the listing `On` or `Off` around a particular section of source.

Literals_in_code — Put literals in code space instead of in data space

Default: **off**

When toggle `Literals_in_code` is On, literals in a program are placed in the code space rather than in the data space. This placement is beneficial if the operating system performs dynamic code loading.

In such circumstances, code is most often read-only, so literals can be swapped out of memory without having to be written to the paging medium used by the operating system.

See Appendix D: *Developing Embedded Applications* for information about ROMable code.

Live_dead_iterate — Perform multiple iterations of live/dead analysis

Default: **off** unless optimization level ≥ 2

When toggle `Live_dead_iterate` is On, the compiler performs multiple iterations of live/dead analysis to further improve the code.

See Chapter 6.5.23: *Live/Dead Analysis (Iterative)* for more information.

Local_CSE_iterate — Perform multiple iterations of local common subexpression elimination

Default: **off** unless optimization level ≥ 2

When toggle `Local_CSE_iterate` is On, the compiler performs multiple iterations of local common subexpression elimination to further improve the code.

See §6.5.9: *Common Subexpression Elimination (Local, Iterative)* for more information.

Long_double_16 — Provide support for 16-byte long double types

Default: **off**

When toggle `Long_double_16` is On, the size of a **long double** type is 16 bytes; when this toggle is **Off**, the size of a **long double** type is 8 bytes.

Long_enums — Map type enum variables to an int-sized area of memory

Default: **off**

When toggle `Long_enums` is On, any variable of an **enum** type is mapped to an **int**-sized region of memory. Otherwise, such a variable is mapped to the smallest of one, two, or four bytes, such that all values are representable.

Long_long_8 — Provide support for eight-byte long long int types

Default: **off**

When toggle `Long_long_8` is **On**, the size of a **long long int** type is eight bytes; when this toggle is **Off**, the size of a **long long int** type is four bytes.

Loop_reg_rename — Perform register lifetime analysis

Default: **off** unless optimization level ≥ 2

When toggle `Loop_reg_rename` is **On**, the compiler performs register lifetime analysis to improve register allocation.

See §6.5.33: *Register Lifetime Analysis* for more information.

Low_level_scheduling — Perform instruction scheduling after register allocation

Default: **off** unless optimization level ≥ 3

When toggle `Low_level_scheduling` is **On**, the compiler performs instruction scheduling after register allocation.

See §6.5.21: *Instruction Scheduling (Low-Level)* for more information.

Make_externs_global — Make objects with storage class extern be global

Default: **on**

When toggle `Make_externs_global` is **On**, any local declaration of an object with storage class **extern** is made global if there is not already a global declaration of the object. Early C compilers promoted an **extern** declaration within a function to the global scope. This toggle supports programs depending upon that feature.

Mem_refs_from_loop — Replace variable memory references in loops with register references

Default: **off** unless optimization level ≥ 2

When toggle `Mem_refs_from_loop` is **On**, the compiler attempts to replace variable memory references in loops with references to registers.

See Chapter 6.5.25: *Loop Memory-Reference Elimination* for more information.

Msgno — Output error message numbersDefault: **On**

By default, High C/C++ compiler warning and error messages are displayed with a three-digit message number. These numbers can also be used as arguments to the `Onwarn`, `Offwarn`, and `Popwarn` pragmas to disable warnings (but not errors) selectively.

Turn `Off` toggle `Msgno` when you do not want the message numbers to be output.

Nested_types — Make nested types invisible outside the class where they are declaredDefault: **On**

C++ only The C++ language includes nested types, meaning that types declared within the scope of a class remain within that scope only and are not visible outside the class.

```
class s { typedef int T; };
T x;      // Error: T is undeclared
```

In pre-2.0 implementations of C++, nested types were made visible to code outside the block in which the nested type was defined.

In 2.0 and later implementations of C++, nested types are not visible to code outside the block.

To accommodate prior practice, turn `Off` toggle `Nested_types`. Turning the toggle `Off` causes any nested types to be declared outside the class as well as inside, in the first enclosing non-class scope. In this example, `T` would be declared within `s` as well as at the global scope, so both `s::T` (the nested-class syntax) and `T` alone (the old syntax) suffice to refer to `T`.

See Chapter 8: *C++-Specific Issues*.

Noalias — Cause optimizer to assume all variables are non-aliasableDefault: **Off** unless optimization level = 7

When toggle `Noalias` is `On`, the optimizer assumes that no variable, pointer-based or otherwise, can be aliased (that is, accessed or modified through any indirect means, such as through a pointer dereference), unless it is explicitly qualified with type qualifier `_Alias`.

WARNING: Turning On toggle `Noalias` for a program that has aliased pointers might cause the generated code to run incorrectly.

For more information, see the discussion of type qualifiers `_Alias` and `_Noalias` in §7.2.2: *Access-Related Type Qualifiers*. Also see §6.2.1: *Specifying an Optimization Level* and §6.5.28: *No Aliasing*.

Nodebug — Do not generate SDI for any named structure within toggle's scope

Default: `Off`

When toggle `Nodebug` is On, the compiler suppresses generation of SDI (symbolic debug information) for all named C and C++ structures within the scope of the toggle, unless toggle `Forceddebug` is also On.

See §13.3: *Minimizing Symbolic Debug Information* for more information.

Note: Toggle `Nodebug` applies only if toggle `Dwarf` is On.

Non_FP_varargs — Do not save floating-point argument registers

Default: `Off`

When toggle `Non_FP_varargs` is On, your application does not save floating-point argument registers. Use this toggle if your application does not expect floating-point values to be passed to its variable-argument functions (for example, if your application is embedded).

Note: Turning On toggle `FP_varargs` implies turning Off toggle `Non_FP_varargs`. However:

- Turning Off toggle `FP_varargs` does not imply turning On toggle `Non_FP_varargs`.
 - Turning toggle `Non_FP_varargs` On or Off has no effect on the status of toggle `FP_varargs`.
-

Caution: The standard MetaWare libraries are compiled with toggle `Non_FP_varargs` turned Off. If you turn this toggle On for code calling library functions that use varargs, you should supply your own libraries.

Optimize_for_space — Produce code that minimizes space

Default: **off** unless optimization level = *s*

When toggle `Optimize_for_space` is On, the code generator produces code sequences that minimize space, sometimes at the expense of execution speed.

Overload_info — Show function chosen during overload operation

Default: **off**

C++ only If toggle `Overload_info` is turned On in C++ compilations, the compiler informs you which function was chosen during overload resolution. This can help if you are not sure at a particular point in your program which function (of a set of overloaded functions) is being called. The information is emitted as a warning.

Parm_warnings — Issue warnings when arguments to a non-prototype function do not match declared parameters

Default: **On**

When toggle `Parm_warnings` is On, the compiler produces warnings whenever arguments to a non-prototype (old-style) function do not match the types of the declared formal parameters of the function. Frequently, this inconsistency is a source of bugs that are disastrous or that are difficult to find.

For example, in this code:

```
double square(x) double x; {return x*x;}
...
printf("%d\n",square(3));
```

`square` is passed the integer 3, not double-precision 3.0, and the compiler issues a warning. The C language definition *prohibits* the compiler from casting 3 to a **double** before passing it.

To eliminate the compiler warnings, turn Off the toggle `Parm_warnings`. We recommend, however, that the program text be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

PCC — Relax ANSI C Standard restrictions to compile PCC programsDefault: **off**

PCC compatible Many UNIX C programs that compile with the AT&T Portable C Compiler do not compile with an ANSI compiler such as High C/C++. When the **PCC** toggle is turned On, many ANSI restrictions are relaxed so that such programs are more likely to compile with High C/C++, with appropriate warnings.

The **PCC** toggle, when On, affects these ANSI Standard C restrictions:

- Structure-member selection is permitted on a variable of any type; it does not need to be a **struct** or **union**.
However, if the name of the member being selected is declared as a member in more than one **struct** or **union** definition, each occurrence must be declared with the same type and be mapped at the same offset.
- Any pointer type is considered compatible with any other pointer type. (Turning On the toggle `Pointers_compatible` also has this effect.)
- Pointers are considered compatible with any integer type. (Turning On the toggle `Pointers_compatible_with_ints` also has this effect.)

Note: To use the unsignedness-preserving semantics employed by PCC, you must turn On toggle `Use_UP_rules`.

These PCC language features are *not* supported:

- old-fashioned assignment operators (for example, `=+`)
- old-fashioned initialization (for example, `int x 0;`)

The **PCC** toggle also makes the inboard macro preprocessor conform more closely to the UNIX C preprocessor.

See also the **-Hpcc** compiler option in Chapter 3: *Using Compiler Options*.

PCC_msgs — Reduce compiler diagnostic capabilities to the PCC levelDefault: **off**

When toggle **PCC_msgs** is On, the compiler's diagnostic capabilities are reduced to the level of the AT&T Portable C Compiler. At this level, the following warnings are not emitted:

```
Function called but not defined.
"=" used where "==" may have been intended.
```

In addition, toggle `PCC_msgs` disables warnings about passing an `int`, for example, to a non-prototyped function expecting a `long int`, because `int` and `long int` are the same size.

When all the warnings are enabled in the High C/C++ compiler, code must be very “clean” to get through the compiler without a warning. Toggle `PCC_msgs` is provided because some developers have code that was designed with a compiler that is not so demanding, and these developers prefer fewer prods from the compiler.

Command-line option `-H+w` turns Off toggle `PCC_msgs`.

`Pointers_compatible` — Assign pointer values to incompatible pointer variables

Default: `off`

When toggle `Pointers_compatible` is On, the compiler permits a pointer value to be assigned to an incompatible pointer variable, with an appropriate warning.

`Pointers_compatible_with_ints` — Make any type pointer compatible with `int` types

Default: `off`

When toggle `Pointers_compatible_with_ints` is On, the compiler allows pointers of any type to be compatible with `int` types, with an appropriate warning.

Although this compatibility is in violation of the ANSI C Standard and High C/C++ specifications, many old C programs improperly assign pointers to `int` types and vice versa. This toggle allows such programs to be compiled without modification.

The ANSI C Standard and High C/C++ disallow this dangerous practice because pointers are not necessarily the same size as `int` types on all machines.

`Postfix_takes_two_args` — Force overloaded postfix operators to require two arguments

Default: `On`

C++ only In C++ compilations, when you overload the postfix operators `++` and `--`, you must use two arguments to distinguish them.

Turn Off toggle `Postfix_takes_two_args` to eliminate this distinction. Then you can define just one `++` operator in a class, and invoke it either with the syntax `++x` or the syntax `x++`.

Preload_args_from_memory — Load function arguments into machine registers in the function prolog

Default: **Off** unless optimization level = 1 or *s*

When toggle `Preload_args_from_memory` is On, the compiler copies function arguments to machine registers in the function prolog to reduce memory fetches.

See §6.5.30: *Preloading Arguments from Memory* for more information.

Caution: Do not turn On toggle `Preload_args_from_memory` when toggle `Global_CSE` is On. If you turn On both toggles, the compiler still generates correct code but might tie up registers unnecessarily.

Print_lattice_members — Include a list of class members in the class lattice display

Default: **Off**

C++ only Turn On toggle `Print_lattice_members` to include a list of class members along with the class-lattice display produced with toggle `Print_lattices`. The members are displayed graphically.

For example, given this program text:

```
class V { int v; };
class A { int a; };
class B : A, virtual V { };
class C : A, virtual V { };
class D : B, C { void f(); };
```

this is the graphical display for class D:

```
D {f}
-----
B           C
-----
A {a} v:V {v} A {a} v:V {v}
```

See toggle `Print_lattices` for an explanation of this format.

Print_lattices — **Generate a graphical view of any class with one or more bases**Default: **off**

C++ only Use toggle `Print_lattices` to obtain a graphical view of any class that has one or more bases. The output is simple character text that can be displayed on any device.

The lattice is printed in two forms.

First form The first form is a one-dimensional linear representation that uses brackets to enclose classes. For example, class `some_class` with bases `b1 ... bn` is displayed as:

```
[some_class b1 ... bn]
```

where `b1 ... bn` is the display of the bases.

Second form The second form is a graphical form. A class `some_class` with bases `b1 ... bn` is displayed as this:

```
some_class
-----
b1      ...      bn
```

where `b1 ... bn` is the display of the bases.

Virtual bases are replicated in the display and are preceded by `v:` to identify them as virtual bases.

For example, given this program text:

```
class V { int v; };
class A { int a; };
class B : A, virtual V { };
class C : A, virtual V { };
class D : B, C { void f(); };
```

the following lattice displays are produced for class D:

```
Class lattice for D at test.cpp, L5/C1, in the
format [D B1 ... Bn]: [D [B A v:V] [C A v:V] ]
```

Graphically, but with duplication of any virtual bases:

```
D
-----
B      C
-----
A v:V A v:V
```

See also toggle `Print_lattice_members`.

Print_layout — Print layout of C++ data structures

Default: **off**

C++ only When toggle `Print_layout` is On, the compiler prints to standard output the layout of all C++ data structures in the module being compiled.

Print_protos — Write a prototype-style function header for every function to standard output

Default: **off**

When toggle `Print_protos` is On, the compiler writes to standard output a new, prototype-style function header for each function definition. This toggle aids in the conversion of C programs to the ANSI Standard C prototype syntax.

For example, for this function definition:

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces the following output:

```
int f(int *x, double (*y)(), int z[]);
```

The old function header can then be replaced with the generated one.

Print_template_usage — Display templates used in a C++ compilation

Default: **off**

C++ only Turn On toggle `Print_template_usage` to find out which templates were used in a C++ compilation. Both function templates and class templates are printed.

See §E.5: *Checking Template Usage: Toggle Print_template_usage*.

Print_var_info — Display mapping of auto-, register-, and argument-class variables

Default: **off**

When toggle `Print_var_info` is On, the compiler displays where each auto-, register-, or argument-class variable is mapped on a per-function basis.

One of the following formats is used:

<code>N ==> R</code>	Specifies that variable <i>N</i> has been mapped to machine register <i>R</i>
<code>N+offset ==> R</code>	Specifies that a member of struct <i>N</i> has been mapped to machine register <i>R</i>
<code>N ==> FP+offset</code>	Specifies that variable <i>N</i> has been mapped into the stack frame at <i>offset</i> from base pointer FP
<code>N ==> <eliminated></code>	Indicates that variable <i>N</i> was eliminated via copy propagation
<code>N ==> <unreferenced></code>	Indicates that variable <i>N</i> was not referenced (or else all references were optimized away); the variable was not mapped

Prolog_trace — Generate information about a function being entered

Default: **off**

When toggle `Prolog_trace` is On, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is entered.

See §13.2: *Tracing Function Calls* for more information.

Prototype_conversion_warn — Warn when a function argument is converted due to a prototype declaration

Default: **off** unless `pragma Warning_level ≥ 2`

When toggle `Prototype_conversion_warn` is On, the compiler generates a warning message when a function's argument is converted due to a prototype declaration.

When you use function prototypes, the compiler can automatically convert a function's argument so that the argument's type matches that of the formal parameter. Wherever such a conversion does not match what would happen in the *absence* of prototypes, such C code would probably not run correctly when compiled with older C compilers that lack prototypes.

Prototype_override_warnings — Warn when an ANSI Standard C declaration overrides an old-style function definition

Default: **On**

When toggle `Prototype_override_warnings` is On, the compiler produces a warning whenever a declaration (not definition) of a function

using the ANSI Standard C prototype syntax overrides the semantics of an old-style function definition. For example:

File header.h:

```
int func(float f, long l);
```

File prog.c:

```
#include "header.h"
int func(f, l) float f; long l; {
    ...
}
void sub() {
    func(3, 4.4); /* Passes 3.0 and 4L via */
}                /* automatic conversion */
```

With the header file included, the interface for `func()` is changed to prototype style (3 is converted to **float** and 4.4 to **long**). If `header.h` were *not* included, the call to `func()` in `sub()` would pass the **int** 3 and the **double** 4.4, and probably `func()` would not work properly. In this example, you can find out how the compiler treats `func()` only by searching all the header files.

We recommend writing function definitions with the ANSI Standard C prototype syntax for improved readability and reliability.

For example:

File prog.c:

*ANSI Standard C
prototype syntax*

```
int func(float f, long l) {
    ...
}
```

You can disable the warning message by turning Off the toggle `Prototype_override_warnings`.

Note: Use toggle `Print_protos` to aid in converting old C programs to the ANSI Standard C prototype syntax.

Read_only_strings — Consider all string constants read-only and store them in the read-only data sectionDefault: **off**

When toggle `Read_only_strings` is On, all string constants are considered read-only and are placed in a read-only data section. Multiple occurrences of identical string literals within a single module appear only once in memory.

Caution: Do not turn On toggle `Read_only_strings` if your program modifies the space occupied by a string literal. For example:

```
static char *p = "a string literal";
...
/* Actually modifies the string literal */
p[0] = 'A';
```

If you compile the preceding program with the toggle `Read_only_strings` turned On, a protection fault or incorrect result might occur at run time.

Recognize_library — Inline code for certain functions in the ANSI Standard C libraryDefault: **On**

When toggle `Recognize_library` is On, the compiler can recognize calls to certain functions defined in the ANSI Standard C library. The compiler might inline the function, perform an arithmetic strength reduction, or treat the call as an operation that is subject to global optimizations.

Candidates for inlining currently include the following math and string functions.

<code>abs()</code>	<code>memcpy()</code>
<code>alloca()</code>	<code>memset()</code>
<code>fabs()</code>	<code>strcmp()</code>
<code>labs()</code>	<code>strcpy()</code>
<code>memcmp()</code>	<code>strlen()</code>

Future releases might encompass more of the library. See §6.5.17: *Inlining ANSI Standard C Functions* for more information.

Reduce_reg_contention — Limit the lifetime of registers to reduce spill code

Default: **off** unless optimization level ≥ 2

When toggle `Reduce_reg_contention` is On, the register allocator attempts to limit the lifetime of registers to reduce spill code.

See §6.5.38: *Spill-Code Reduction* for more information.

Reference_all_functions — Create a reference to every non-inline function in the program

Default: **off**

C++ only Turn On toggle `Reference_all_functions` to create a reference to every non-inline function declared in your program. The reference is made within the function `main()`. The result is not intended to be executed, but only linked. Successful linking means that each such function has been implemented.

Use toggle `Reference_all_functions` to check that you have implemented all functions that you declared in, for example, a class library.

Reg_alloc_enhance — Enhance register allocation

Default: **off** unless optimization level ≥ 2

When toggle `Reg_alloc_enhance` is On, the compiler attempts to improve register allocation.

See §6.5.31: *Register Allocation (Enhanced)* for more information.

Relative_includes — Look up included files relative to the directory of the including file

Default: **On**

When toggle `Relative_includes` is On, the compiler looks up the *filename* of

```
#include "filename"
```

relative to the directory of the file containing the `#include`. This is UNIX C standard practice; it permits compilation of a source file that is in a directory other than the current working directory.

When toggle `Relative_includes` is Off, the compiler looks up *filename* in the current working directory.

RTTI_common — Solve the single-copy problem for RTTI structuresDefault: **On**

C++ only When toggle `RTTI_common` is **On**, it solves the single-copy problem for classes without seats declared within its scope, or for non-class types.

For a discussion of the single-copy problem and its solutions, see §8.8.1: *The Single-Copy Problem*.

When you use option `-Hnocomdat`, this toggle is disabled, turned off.

Run_time_type_info — Generate and store run-time type information (RTTI) for a classDefault: **off**

C++ only Toggle `Run_time_type_info` ensures that, for every class compiled within its scope, run-time type information (RTTI) is generated and stored in the class's virtual function tables.

If toggle `Run_time_type_info` is not turned **On**, RTTI is generated only as needed by `typeid`, and there is no guarantee that the virtual-function tables will contain a pointer to RTTI.

See the **High C/C++ Language Reference** and §8.5: *Generating Run-Time Type Information (RTTI)* for more information on RTTI.

Save_incoming_regs — Save all incoming registers of a functionDefault: **off**

When toggle `Save_incoming_regs` is **On**, all incoming registers of a called function are saved in the function's argument save area. (Incoming registers are the argument registers used to pass arguments to the called function.)

See §11.5.6: *Tag Support* for more information about tags.

Single_static_assignment — Place intermediate representation of code in single static assignment formDefault: **off** unless optimization level ≥ 5

When toggle `Single_static_assignment` is **On**, the compiler puts the intermediate representation of the code being compiled in single static assignment form, so the optimizer can perform additional global copy, constant propagation, and other optimizations.

single_math_lib — **Call ANSI Standard C math functions in single precision if possible**

Default: **On** unless you specify command-line option **-HL** (little-endian)

When toggle `single_math_lib` is **On**, calls to some ANSI Standard C math functions are faster: the compiler makes single-precision calls if the arguments and return-value types match the prototypes of the math functions called.

source_code_order — **Produce code in source order where possible**

Default: **Off** unless optimization level = 0 or 1

When toggle `source_code_order` is **On**, the compiler attempts to generate code in source order. This makes code easier to debug at the assembly-language level.

Note: Toggle `source_code_order` is turned **On** when you specify option **-g**.

For more information, see §6.5.5: *Code in Source-Code Order*.

strength_reduce — **Perform loop strength reduction**

Default: **Off** unless optimization level ≥ 5

When toggle `strength_reduce` is **On**, the compiler performs loop strength reduction.

See §6.5.26: *Loop Strength Reduction* for more information.

strict_ANSI_math — **Generate strict ANSI C Standard-conforming code for standard-math function calls**

Default: **Off** unless you specify option **-Hansi**

Turn **On** toggle `strict_ANSI_math` to generate strictly ANSI C Standard-conforming code for calling standard math functions (functions defined in the `math.h` file). The compiler then generates calls to library functions for the transcendentals instead of inlining them. The called functions set `errno` to indicate existing error conditions.

Without the calls, the compiler inlines many transcendentals; after the Common Subexpression Elimination phase of optimization, `errno` might not always be set.

If you specify option **-Hansi** and would like to get faster transcendentals, but are not concerned about `errno` values, you can turn Off toggle `Strict_ANSI_math`.

See also toggle `Recognize_library`.

Tag_table — **Generate tag table in separate .tag section for each function**

Default: **off**

When toggle `Tag_table` is On, the compiler generates a tag table in a separate `.tag` section for each function. The tag table is used for debugging information.

See also toggle `Save_incoming_regs`.

Note: Toggles `Tag_table` and `Tag_table_in_text` are mutually exclusive. Using both results in undefined behavior.

See §11.5.6: *Tag Support* for more information about tags.

Tag_table_in_text — **Generate tag table in .text section for each function**

Default: **off**

When toggle `Tag_table_in_text` is On, the compiler generates a tag table in the `.text` section for each function. The tag table is used for debugging information.

See also toggle `Save_incoming_regs`.

See §11.5.6: *Tag Support* for more information about tags.

Tail_recursion — **Optimize recursive functions where possible**

Default: **off**

When toggle `Tail_recursion` is On, the compiler attempts to optimize recursive functions.

For more information, see §6.5.39: *Tail Recursion*.

Template_common — **Solve the single-copy problem for template functions and classes**

Default: **On**

C++ only When toggle `Template_common` is On, it solves the single-copy problem for all virtual function tables, RTTI structures, inline functions, and

out-of-line function bodies generated as a result of instantiating a template that was declared within the scope of this toggle.

For a discussion of the single-copy problem and its solutions, see §8.8.1: *The Single-Copy Problem*.

Template_trivial_conversions — Allow trivial conversions for matching arguments to function templates

Default: **off**

C++ only In C++ compilations, turn On toggle `Template_trivial_conversions` to permit so-called trivial conversions (as defined in the **Annotated C++ Reference Manual**) to be allowed when matching arguments to function templates.

See also §8.8.3: *Problems with Matching const Parameters*.

Thread — Specify that variables are to be stored in thread-local storage

Default: **off**

When toggle `Thread` is On, all data declared or defined within its scope is stored in thread-local storage. Toggle `Thread` is the EasyThread™ alternative for `__declspec(thread)`.

Trap_div_by_zero — Trap division by zero

Default: **off**

When toggle `Trap_div_by_zero` is On, the compiler generates code to check every integer divide operation for division by 0 (zero). This code contains a trap that is triggered at run time if a divide-by-zero occurs.

Trigraphs — Enable trigraphs

Default: **off** unless command-line option **-Hansi** is specified

When toggle `Trigraphs` is On, the compiler recognizes *trigraphs* — sequences of three characters that are converted to a single character.

Trigraphs are useful on machines that have limited character sets; they allow you to represent characters that would otherwise be unavailable.

Note: On ASCII-based systems, trigraphs are not generally useful. All the characters that trigraphs represent are typically available on contemporary keyboards.

Trigraphs are required for ANSI Standard C compatibility, so toggle `Trigraphs` is turned On by the `-Hansi` option.

Try_static_cast — Replace old-style type casts with static_cast

Default: `Off`

C++ only Toggle `Try_static_cast` causes all old-style C++ casts within its scope of the form

`(Type)expression`

to be treated as the new type cast `static_cast`. This toggle also causes the compiler to issue a warning at each line containing a cast.

For more information about the new C++ cast notation, see the **High C/C++ Language Reference** and §8.6: *Switching to New-Style Type Cast Notation: Toggle `Try_static_cast`*.

Use_eieio — Enforce in-order execution of input/output

Default: `Off`

When toggle `Use_eieio` is `Off`, the compiler does not generate `eieio` instructions for memory stores of volatile variables. When toggle `Use_eieio` is `On`, the compiler generates `eieio` instructions for all memory stores of volatile variables.

Refer to §6.5.19: *In-Order Execution of I/O (Disable)* for more information about using toggle `Use_eieio`.

Use_powerpc_mnemonics — Use PowerPC assembler mnemonics

Default: `On`

When toggle `Use_powerpc_mnemonics` is `On`, the compiler uses PowerPC assembler mnemonics. When the toggle is `Off`, the compiler uses POWER assembler mnemonics.

Use_reciprocal_for_divide — Convert a floating-point divide-by-constant to a multiply-by-reciprocalDefault: **off**

When toggle `Use_reciprocal_for_divide` is On, a floating-point divide by a constant is converted to a multiply by the reciprocal of the constant, because a multiply is typically faster than a divide.

Note: Even if toggle `Use_reciprocal_for_divide` is Off, the compiler might choose to multiply by the reciprocal if the reciprocal has an exact binary representation; for example, the reciprocal of 2.

Caution: This conversion can introduce inaccuracy into a program, because the reciprocal might not be exact. In general, the last bit of the result might be different. For most programs, this does not matter; toggle `Use_reciprocal_for_divide` is provided for users who want the extra performance.

Use_UP_rules — Widen unsigned char and unsigned short to unsigned intDefault: **off**

In C, there are two rules for implicitly widening an **unsigned char** or **unsigned short** operand in an arithmetic expression: *value-preserving* (VP) and *unsignedness-preserving* (UP).

Value preserving The VP rule widens an **unsigned char** or **unsigned short** to **signed int**. This is the rule used in ANSI Standard C.

Unsignedness preserving The UP rule widens such operands to **unsigned int**. This rule is used by older compilers, such as those based on the AT&T Portable C Compiler.

Some programs behave differently depending on which rule is in force. For example:

```
unsigned char c;
...
if (c-1 > 0) fnc();
```

If the value of `c` is 0 (zero) when the `if` statement is executed, the relational expression is false if the VP rule is used, true if the UP rule is used.

This latter characteristic occurs because `c` is widened to an **unsigned int**; therefore, the type of the expression `c-1` is also **unsigned int**.

If you are compiling code that was originally developed under a UP compiler, turn On toggle `Use_UP_rules`.

See the description of toggle `VP_UP_warn`.

Caution: If toggle `Use_UP_rules` is turned On, some macros in the standard header files might not function properly, because they were written with the assumption that value-preserving rules apply.

VP_UP_warn — **Warn about expressions with more than one value, depending on whether UP or VP rules are used**

Default: **Off** unless pragma `Warning_level = 4`

When toggle `VP_UP_warn` is On, the compiler issues a warning when it encounters an expression that could have more than one value, depending on whether value-preserving or unsignedness-preserving rules are used.

See the description of toggle `Use_UP_rules`.

Vtable_common — **Solve the single-copy problem for virtual function tables**

Default: **On**

C++ only When toggle `Vtable_common` is On, it solves the single-copy problem for virtual function tables, for classes without a seat declared within the scope of the toggle.

For a discussion of the single-copy problem and its solutions, see §8.8.1: *The Single-Copy Problem*.

Widen_float_args — **Widen float arguments to double**

Default: **On** unless you specify option **-fsingle2**

The compiler ordinarily widens arguments of type **float** to type **double** under the following conditions:

- the argument is passed to a non-prototyped function
- the argument is passed as a variable argument to a prototyped function

(In both cases, the compiler widens the argument *before* passing it.) This is the ANSI Standard C convention.

When toggle `widen_float_args` is turned `Off`, no such conversions are performed.

WARNING: Beware of passing a `float` argument to a function that was compiled with a different setting for toggle `widen_float_args`. For example, when this toggle is `Off`, the values of `float` expressions cannot be displayed with `printf()`.
