

HP C/aC++ Version 6 Features to Improve Developer Productivity

Version 1.5



Introduction	4
What's New	4
Compile Time Diagnostics.....	4
Security Diagnostics	7
Customization of Compiler Diagnostics	9
Runtime Checking	11
Execution Path Recovery	14
Further information.....	16

Introduction

The most time consuming and expensive task in software development is to find and fix defects. The later a defect is found in the software development process, the costlier it gets. A bug found and fixed at the coding or testing stages is better and less expensive than the same found by a user after the product release. The HP compilers for Integrity provide a number of features to help software developers detect potential problems in their programs earlier in the process.

Compile time diagnostics and Run time checks are provided to allow the developer to identify many common problems before the code is checked in. In addition, the compiler supports enhanced debugging capabilities. Execution Path Recovery feature of the compiler can provide backtracking of the execution path from the crash point in a core file or from a breakpoint in the debugger

All of the above features of the compiler lower maintenance and support costs and free up developers to spend more time implementing new features.

What's New

The A.06.15 release of the HP C and C++ compilers provides `+wlock`, `+wperfadvice`, `+Wmacro` and `+check=bounds:pointer|globals|truncate` features, in addition to improvements on the features provided by older versions. This version also introduces the execution path recovery (`+pathtrace`) feature to backtrack the execution path from the crash point in a core file or from a breakpoint in the debugger.

The A.06.10 release of the HP C and C++ compilers provides `+wendian` option and improvements to the existing developer productivity features. This release also introduces a new C/C++ static analysis tool, HP Code Advisor, which provides all the compile time diagnostic features of Integrity compilers. This tool is available free of cost on HP-UX Integrity and PA-RISC. For more information, visit the [HP Code Advisor \(cadvise\)](#) homepage.

The compiler features to improve developer productivity are available starting with the A.06.05 release of the HP C and C++ compilers. The A.06.05 release provides `+wlint`, `+w64bit`, `+wsecurity` and `+check=all|none|bounds|malloc|stack|uninit` options.

The Version 6 of HP C and C++ compilers provide a rich set of diagnostics that are available by default and with `+w`, `+Oinfo` and `+Oinitcheck` options.

Compile Time Diagnostics

The HP compilers contain numerous checks for potential correctness problems. Some diagnostics are emitted by default while others are enabled via explicit options listed below:

+wlint

This option enables several warnings in the compiler that are similar to the functionality of lint check. These compile time diagnostics can be very useful in detecting potential problems in the source code. The number of warnings with this option may be up to 5-10 times more than those emitted by default by the compiler.

Example

```
$ cat bitfield.c
struct { int bit:1; } s;

void test()
{
    s.bit = 1;
}

$ cc -c +wlint bitfield.c
"bitfield.c", line 1: warning #2108-D: signed bit field of
length 1
    struct { int bit:1; } s;
                ^

"bitfield.c", line 5: warning #4251-D: the assignment has an
excessive size for a bit field
    s.bit = 1;
            ^
```

Other notable examples of warnings enabled with the +wlint option:

- Argument is incompatible with formal parameter
- Function declared implicitly
- Function is re-declared after being called
- Type conversion may truncate value
- Unsigned value cannot be less than zero
- Missing return statement at end of non-void function
- Nested comment is not allowed
- Signed bitfield of length 1
- Memory leak
- Potential null pointer dereference
- Detection of uncaught exceptions
- Uninitialized variables

+w64bit

This option enables warnings that help you detect potential problems in converting 32-bit applications to 64-bit. The +w64bit option applies both to 32 bit and 64 bit compilers. It is equivalent to the +M2 option.

Some of the checks performed are:

- 64bit value is implicitly converted to a 32bit value, e.g. long to int.
- Pointer to 4-byte aligned object implicitly converted to a pointer to 8-byte aligned object.

Example

```
$ cat convert.c
int *int_to_ptr (int i)
{
    return (int *)i;
}

$ cc -c +w64bit convert.c
"convert.c", line 3: warning #4231-D: 64 bit migration:
conversion between types of different sizes has occurred
(from "int" to "int * ")
    return (int *)i;
            ^
```

+wendian

This option enables diagnostics that identify areas in the source code that might have porting issues between little-endian and big-endian. For example

- de-reference which could cause endian-dependent behavior
- Union definition that is endian dependent

Example

```
$ cat endian.c
union ul {
    char c[4];
    int v;
};
$ cc -c +wendian endian.c
"endian.c", line 1: warning #4289-D: endian porting: the
definition of the union may be endian dependent
union ul {
    ^
```

+wlock

This option detects multi-threaded programming issues and enables diagnostics for potential errors in using lock/unlock calls in multi-threaded programs that use the pthread library. This is based on cross-module analysis performed by the compiler, which is much more powerful compared to simple scanning and parsing tools. The `+wlock` option implicitly enables a limited form of cross-module analysis even if `-ipo` or `+O4` options are not specified. This may lead to a significant increase in the compile time in comparison to a build without the `+wlock` option. Using this option may result in the compiler invoking optimizations other than those which are part of the specified optimization level. If `+wlock` is used with `-ipo` or `+O4` option, the generated code is not affected and the compile time does not increase much.

The problems detected by `+wlock` diagnostics include

- acquiring an already acquired lock
- releasing an already released lock
- unconditionally releasing a conditionally acquired lock

Example

```
$ cat lock.c
#include <pthread.h>
#include <stdio.h>

int a;
pthread_mutex_t Mutex;

void perform_operation(pthread_mutex_t* mutex1, int
increment, int* global) {

    if (increment > 10){
        int status = pthread_mutex_lock(mutex1);
    }
    *global = *global + increment;
    int status = pthread_mutex_unlock(&Mutex);
}

int main(void) {
    int i;
    scanf("%d", &i);
    perform_operation(&Mutex, i, &a);
    printf("%d is value\n", a);
}
```

```
$ cc +wlock lock.c
"lock.c", line 12: warning #20223-D: Trying to unlock a
lock held conditionally
```

+wperfadvice

This option enables performance advisory messages. It offers both integrity-specific and architecture-independent performance advice. The advice emitted is dependent on the optimization options used for compilation.

Example

```
$ cat large.c
struct X{
    int i;
    int arr[1000];
} x;

int foo( struct X);
int main() {
    foo (x);
}

$ cc -c +wperfadvice large.c
"large.c", line 8: warning #4319-D: performance
advice: passing a
large (4004 bytes) parameter by value is
inefficient, consider passing
by reference
foo (x);
    ^
```

+w

Enable all warnings about potentially questionable constructs in the compiler. This includes warnings such as `+wlint` and `+w64bit` warnings. The number of warnings with this option may be up to 5-10 times more than those emitted with `+wlint`. Some examples of warnings enabled with `+w` option:

- Variable is declared but never referenced
- Comparison of unsigned integer with signed integer
- Padding size of structure to alignment boundary
- Argument is incompatible with corresponding format string conversion

+Oinfo

This option displays informational messages about the optimization process. It may be helpful in understanding what optimizations are occurring. These messages are not emitted with the `+w` option.

+Oinitcheck

This option enables warnings about local variables that may be used before they are defined. Many of the warnings generated with this option may be false positives.

Security Diagnostics

+wsecurity

This option enables compile time diagnostics for potential security vulnerabilities. Security flaws are not only very costly to fix, but also can lead to a potential loss of customers and reputation. Most developers are not trained to detect security vulnerabilities.

With the `+wsecurity` option, warnings are emitted for scenarios where untrusted (tainted) data may reach a critical reference point in the program. This is based on cross-module analysis performed by the compiler, which is much more powerful compared to simple scanning and parsing tools. The `+wsecurity` option implicitly enables a limited form of cross module analysis, even if `-ipo` or `+O4` options are not specified. This may lead to a significant increase in the compile time in comparison to a build without the `+wsecurity` option. Using this option may cause the compiler to invoke optimizations other than those which are part of the user-specified optimization level. If `+wsecurity` is used in addition to `-ipo` or `+O4`, the generated code is not affected and the compile time does not increase much.

The problems detected include use of unsafe APIs, use of unsafe data length argument, unsafe loop exit condition, unsafe file path use, etc.

1. Example of reference to untrusted file path:

```
% cat popen.c
#include <stdio.h>
#include <stdlib.h>

char* get_path()
{
    return getenv("BLAHBLAH");
}

int main()
{
    char* path = get_path(); // line 11

    FILE* my_pipe = popen(path, "r"); // line 13
    printf ("%p\n", my_pipe);
}
% cc +wsecurity popen.c
"popen.c", line 13, procedure main: warning #20116-D: (SECURITY) Tainted
value may be used as path or file name
++ tainted value is returned from 'get_path' called by 'main' at line 11
in file popen.c
```

2. Example of unsafe loop exit condition:

```
int a[100];
int loop(int i)
{
    for (int j = 0 ; j < i; j++) // line 5
        a[j] = 0;
    return a[0];
}

int main()
{
    int i;
    fread(&i, 1,4,stdin);
    loop(i);
}

"loop1.c", line 5, procedure loop: warning #20114-D: (SECURITY) Tainted
value may be used in loop exit condition computation
++ 'loop' is called by 'main' at line 14 in file loop1.c
++++ Tainted value is obtained from 'main'
```

Customization of Compiler Diagnostics

Compiler Diagnostics have the following conflicting goals:

- Emit all possible messages from the compiler to ensure that the user gets the maximum information about potential problems detected by the compiler.
- Reduce the number of messages so that the user is not overwhelmed by the sheer magnitude of the warnings.
- Eliminate or reduce the number of benign or misleading messages

The right balance for these conflicting goals is different for each specific situation. It varies on the basis of various criteria such as the size of the application, resources allotted to eliminate warnings by changing source code, total number of messages emitted, new code or port of existing code, and coding guidelines. The recommended model is to use options such as `+wlint`, `+w64bit`, and `+w` to enable the right level of warnings and then use options to disable specific warnings that are not of interest or are too noisy.

Each distinct type of diagnostic emitted by the compiler has a number associated with it. The user can control the emission of each diagnostic separately. This allows the user to focus on specific warnings by choice. The following options and pragmas allow warnings to be disabled, enabled or considered as errors:

+Warg1[,arg2,...,argn]

This option selectively suppresses any specified warning messages. `arg1` through `argn` are valid compiler warning message numbers.

+Wwarg1[,arg2,..,argn]

This option selectively enables emission of compiler diagnostics that are not enabled by default. `arg1` through `argn` are valid compiler warning message numbers.

+Wearg1[,arg2,..,argn]

This option selectively interprets any specified warning messages as errors. `arg1` through `argn` must be valid compiler warning message numbers. This allows the user to enforce a policy to disallow specific warnings by forcing an error at compile time.

Conflicts between `+w`, `+ww` and `+we` options are resolved based on their severity. `+we` is the highest and `+w` is the lowest.

+Wmacro:MACRONAME:d1,d2,...,dn

This option disables warning diagnostics `d1`, `d2`, . . . , `dn` in the expansion of macro `MACRONAME`. If `-1` is given as the warning number, then all warnings are suppressed. This option is not applicable to warning numbers greater than 20000. `+Wmacro` has higher priority than the other diagnostic-control command-line options that are applicable to the whole source. Diagnostic control pragmas take priority based on where they are placed.

+opts filename

All compiler options, can be consolidated in a single configuration file using the +opts option. This reduces the clutter on the command line and provides a single location to specify the customizations. Comment lines can also be inserted in the +opts configuration file.

Example

```
$ cat warnings_config
# change warning about use of undefined variable to error
+We2549

# enable warning about redeclaration of variable
+Ww3348

# disable warning: statement is unreachable
+W2111

$ cc +opts warnings_config +wlint -c uninit.c
"uninit.c", line 6: warning #3348-D: declaration hides variable "i"
(declared at line 3)
    int i;
      ^

"uninit.c", line 9: error #2549-D: variable "i" is used before its value
is set
    if (i) j = 3;
      ^

1 error detected in the compilation of "uninit.c".
```

#pragma diag_suppress|diag_warning|diag_error NNNN

#pragma diag_default NNNN

Command line options provide control of diagnostics emission for the entire build or for a specific source file. There are several pragmas which allow the user to control warnings for a specific region within a source file. The use of #pragma diag_suppress disables generation of warning N after the pragma in the source file. The pragma diag_default restores the default handling for diagnostic N. Similarly, diag_warning enables emission of a diagnostic and diag_error converts a warning to an error.

Runtime Checking

In addition to compile time diagnostics, the HP compilers provide several different `+check` options to detect some types of errors at runtime.

Advantages of runtime checks in comparison to compile time diagnostics

- Failed runtime checks always indicate a real problem. There are no false positives.
- Can help detect several hard to catch defects like stack overflow or out-of-bounds accesses that may not be possible with compile time diagnostics.
- Developer does not have to analyze and fix warnings. Action needs to be taken only when a runtime check fails.

Disadvantages of runtime checks

- Runtime checks slow down the user program due to additional instrumentation that is added. In general, they should be used only during development (debug builds) and not for released software (production builds).
- Runtime checks do not cover all paths in the application. Compile time diagnostics can analyze and cover all paths in the source.
- Compile time diagnostics detect problems earlier and with less overhead.

`+check=all | none | bounds | globals | malloc | stack | truncate | uninit`

The `+check` option provides runtime checks to detect many common coding errors in the user program. These options introduce additional instructions for runtime checks that may significantly slow down the user program. A failed check will result in the program aborting at runtime. In most cases, an error message and a stack trace will be emitted to `stderr` before program termination. The `+check` options need to be specified at both compile time and link time since they may require additional libraries to be linked into the user program. If different `+check` options are specified while compiling different source files, all the specified `+check` options are needed at link time.

A failed runtime check causes the program to abort. Set the environment variable `RTC_NO_ABORT` as `1` to continue the execution and find any further runtime failures. This will allow emission of messages for all failed runtime checks. The program abort is deferred till the exit.

Multiple `+check` options are interpreted left to right. In case of conflicting options, the one on the right will override an earlier `+check` option.

`+check=all`

This option enables all runtime checks provided by the compiler. It overrides any `+check=x` options that appear earlier on the command line. The `+check=all` option is currently equivalent to the following options:

```
+check=bounds
+check=malloc
+check=stack:variables
+check=uninit -z
+Olit=all
```

The `-z` or `+Olit=all` options that are part of `+check=all`, can be overridden by an explicit `-z` or `+Olit=none` option.

+check=none

Turn off all runtime checking options. It disables any `+check=x` options that appear earlier on the command line.

+check=bounds[:array | pointer | all | none]

The `+check=bounds` option enables checks for out-of-bounds references to array variables or to buffers through pointer access. The check is performed for each reference to an array element or pointer access. If a check fails, an error message is emitted and the program is aborted.

+check=bounds:array

This option enables check for out-of-bounds references to array variables. It applies only to local and global array variables. It also applies to references to array fields of `struct`. It does not apply to arrays allocated dynamically using `malloc` or `alloca`.

+check=bounds:pointer

This option enables check for out-of-bounds references to buffers through pointer access. The buffer could be a heap object, global variable, or local variable. This sub option also checks out-of-bounds access through common `libc` function calls such as `strcpy`, `strcat`, `memset`, and so on. This check can create significant run-time performance overhead.

+check=bounds:all

This option enables out-of-bounds checks for both arrays and pointers.

+check=bounds:none

This option disables out-of-bounds checks.

+check=bounds (with no suboption) is equal to `+check=bounds:array`

Note:

This may change in the future to also include `+check=bounds:pointer`.

+check=all enables `+check=bounds:array` only. You can combine `+check=bounds:[pointer|all]` with all other `+check` options, except `+check=globals` (which would be ignored in this case).

See the `+check=malloc` and the `+check=stack` options for related runtime checks for heap and stack objects.

+check=globals

This option enables runtime checks to detect corruption of global variables, by introducing and checking "guards" between them, at the time of program exit. Setting environment variable `RTC_ROUTINE_LEVEL_CHECK` will also enable the check whenever a function compiled with this option returns.

For this purpose, the definition of *global* is extended to be all variables that have static storage duration, including file or namespace scope variables, function scope static variables, and class (or template class) static data members.

+check=stack[:frame | :variables | :none]

This option enables runtime checks to detect writes outside stack boundaries. Markers are placed before and after the whole stack frame and around some stack variables. On procedure exit, a check is done to see if any marker has been overwritten. If any stack check fails, an error message and stack trace is written to `stderr` and the program is aborted. The stack checks are not performed for an abnormal exit from the procedure (e.g. using `longjmp()`, `exit()`, `abort()` or exception handling).

+check=stack:frame

This option enables runtime checks for illegal writes from the current stack frame that overflow into the previous stack frame.

+check=stack:variables

This option enables runtime checks for illegal writes to the stack just before or after some variables on the stack. This includes array, struct/class/union and variables whose address is taken. It also includes the overflow check for the stack frame (+check=stack:frame). In addition to the above checks, this option causes the whole stack to be initialized to a "poison" value, which can help detect the use of uninitialized variables on the stack.

+check=stack:none

This option disables all runtime checks for the stack.

+check=stack

The +check=stack option without any qualifiers is equivalent to +check=stack:variables at optimization levels 0 and 1. It is equivalent to +check=stack:frame for optimization level 2 and above.

+check=truncate[:explicit|:implicit]

The +check=truncate[:explicit|:implicit] option enables runtime checks to detect data loss in assignment when integral values are truncated. Data loss occurs if the truncated bits are not all the same as the left most non-truncated bit for signed type or not all zero for unsigned type.

Programs might contain intentional truncation at runtime, such as when obtaining a hash value from a pointer or integer. To avoid runtime failures on these truncations, you can explicitly mask off the value:

```
ch = (int_val & 0xff);
```

Note that the +check=all option does not imply +check=truncate. To enable +check=truncate, you must explicitly specify it.

+check=truncate:explicit

This option turns on runtime checks for truncation on explicit user casts of integral values, such as (char)int_val.

+check=truncate:implicit

This option turns on runtime checks for truncation on compiler-generated implicit type conversions, such as ch = int_val.

+check=truncate

This option turns on runtime checks for both explicit cast and implicit conversion truncation.

+check=uninit

This option checks for a use of a stack variable before it is defined. If such a use is detected, an error message is emitted and the program is aborted. The check is done by adding an internal flag variable to track the definition and use of user variables. See the +Oinitcheck option to enable compile time warnings for variables that may be used before they are set.

+check=malloc

This option enables memory leak and heap corruption checks at runtime. It will cause the user program to abort for writes beyond boundaries of heap objects, free or realloc calls for a pointer that is not a valid heap object, and out-of-memory conditions. Memory leak information is captured and written to a log file when the program exits. The name of the logfile is printed before program termination.

The `+check=malloc` option works by intercepting all heap allocation and deallocation calls. This is done by the use of a debug malloc library, `librtc.so`. The option works for programs that use the system `malloc` or for user-provided `malloc` routines in a shared library. The `librtc.so` library is also used by the HP WDB debugger to provide heap memory checking features in the debugger. Please refer to the HP WDB debugger documentation for more information about heap memory checking. The `librtc.so` library is shipped as part of the `wdb` product.

Install the HP WDB bundled with the compiler or a more recent version of `wdb` to get full functionality. The default behavior of the `+check=malloc` option can be changed by users providing their own `rtcconfig` file.

The user specified `rtcconfig` file can be in the current directory OR in a directory specified by the `GDBRTC_CONFIG` environment variable. The default configuration used by the `+check=malloc` option is:

```
check_bounds=1; check_free=1; scramble_block=1; abort_on_bounds=1;
abort_on_bad_free=1; abort_on_nomem=1; check_leaks=1; min_leak_size=0;
check_heap=0; frame_count=4; output_dir=.
```

For a description for the above configuration parameters and the full list of other parameters, refer to the HP WDB debugger documentation.

Execution Path Recovery

This feature is enabled by the **+pathtrace[=local|global|global_fixed_size|none]** Option. It provides a mechanism to record program execution control flow into global and/or local path tables. The saved information can be used by the HP WDB debugger to assist with crash path recovery from the core file, or to assist in debugging the program by showing the executed branches.

Currently only `if-else`, `switch-case-default`, and `try-catch` execution paths are recorded in the path table. If there is no condition statement inside `for`, `while`, or `do-while` loop, then no execution path is recorded.

+pathtrace=local

This option generates a local path table and records basic block-execution information in it at runtime.

+pathtrace=global

This option generates a global path table and records basic block-execution information in it at runtime.

+pathtrace=global_fixed_size

This option generates a fixed-size (65536 items) global path table and records basic block-execution information in it at runtime. This form differs from `+pathtrace=global` because the size of the table cannot be configured at runtime, and the contents cannot be dumped to a file. The fixed-size global path table has better runtime performance than the configurable global path table. The performance difference varies depending on the optimization level and how the program is written.

+pathtrace=none

This option disables generation of both the global and local path tables.

The values can be combined by joining them with a colon. For example:

`+pathtrace=global:local`. The `global_fixed_size` and `global` values are mutually exclusive. If more than one of them is specified on the command line, the last one takes precedence. The same is true for the `none` value.

`+pathtrace` with no values is equivalent to `+pathtrace=global_fixed_size:local`

The use of this option and the `-mt` option must be consistent for all compilation and link steps. That means if `-mt` is used with `+pathtrace` at compile time, it should also be used at link time; if `-mt` is not used with `+pathtrace` at compile time, it should not be used at link time.

For information on how to view the execution paths in the debugger refer to the "Printing the Execution Path Entries for the Current Frame or Thread" feature in HP WDB documentation.

Further information

For more information refer www.hp.com/go/cadvise