

HP Code Advisor Diagnostics Reference Guide

HP Part Number: 5900-1865
Published: July 2011
Edition: 4



© Copyright 2011 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein. UNIX is a registered trademark of The Open Group.

Contents

About This Document.....	7
Intended Audience.....	7
HP Encourages Your Comments.....	7
1 Diagnostics Categorization.....	8
Default Diagnostics.....	8
+wall Diagnostics.....	16
+wlint Diagnostics.....	16
+w64bit Diagnostics.....	18
+wendian Diagnostics.....	19
+wsecurity Diagnostics.....	19
+wlock Diagnostics.....	20
+wperfadvice Diagnostics.....	20
+w/+w1 Diagnostics.....	21
Diagnostic messages turned off by default.....	22
2 Diagnostics Details.....	24
2009 nested comment is not allowed.....	24
2028 expression must have a constant value.....	24
2042 operand types are incompatible (%t1 and %t2).....	24
2047 incompatible redefinition of macro %nod.....	25
2054 Too few arguments in macro invocation.....	25
2063 shift count is too large.....	25
2064 declaration does not declare anything.....	26
2068 integer conversion resulted in a change of sign.....	26
2069 integer conversion resulted in truncation.....	26
2077 this declaration has no storage class or type specifier.....	26
2082 storage class is not first.....	27
2102 forward declaration of enum type is nonstandard.....	27
2108 signed bit field of length 1.....	27
2111 statement is unreachable.....	28
2117 non-void %n should return a value.....	28
2120 return value type does not match the function type.....	28
2128 loop is not reachable from preceding code.....	28
2144 a value of type %t1 cannot be used to initialize an entity of type %t2.....	29
2147 declaration is incompatible with %nfd.....	29
2152 conversion of nonzero integer to pointer.....	29
2167 argument of type %t1 is incompatible with parameter of type %t2.....	30
2170 pointer points outside of underlying object.....	30
2172 external/internal linkage conflict with previous declaration %nod.....	30
2174 expression has no effect.....	30
2175 subscript out of range.....	31
2177 %n was declared but never referenced.....	31
2180 argument is incompatible with formal parameter.....	31
2181 argument is incompatible with corresponding format string conversion.....	32
2186 pointless comparison of unsigned integer with zero.....	32
2187 use of "=" where "==" may have been intended.....	32
2188 enumerated type mixed with another type.....	32
2191 type qualifier is meaningless on cast type.....	33
2192 unrecognized character escape sequence.....	33
2193 zero used for undefined preprocessing identifier.....	33
2223 function %sq declared implicitly.....	33

2224	the format string requires additional arguments.....	34
2225	the format string ends before this argument.....	34
2226	invalid format string conversion.....	34
2228	trailing comma is nonstandard.....	34
2231	declaration is not visible outside of function.....	35
2236	controlling expression is constant	35
2237	selector expression is constant.....	35
2245	a nonstatic member reference must be relative to a specific object.....	35
2248	pointer to reference is not allowed.....	36
2249	reference to reference is not allowed.....	36
2250	reference to void is not allowed.....	36
2251	array of reference is not allowed.....	36
2252	reference %n requires an initializer.....	37
2260	explicit type is missing ("int" assumed).....	37
2263	duplicate base class name.....	37
2265	%nd is inaccessible.....	37
2267	old-style parameter list (anachronism).....	38
2269	conversion to inaccessible base class %t is not allowed.....	38
2276	name followed by :: must be a class or namespace name.....	38
2278	a constructor or destructor may not return a value.....	39
2306	default argument not at end of parameter list.....	39
2314	only nonstatic member functions may be virtual.....	39
2315	the object has cv-qualifiers that are not compatible with the member function.....	39
2319	pure specifier ("= 0") allowed only on virtual functions.....	40
2320	badly-formed pure specifier (only "= 0" is allowed).....	40
2321	data member initializer is not allowed.....	40
2322	object of abstract class type %t is not allowed.....	41
2323	function returning abstract class %t is not allowed.....	41
2325	inline specifier allowed on function declarations only.....	41
2329	local class member %n requires a definition.....	42
2336	unknown external linkage specification.....	42
2340	value copied to temporary, reference to temporary used.....	42
2363	invalid anonymous union - nonpublic member is not allowed.....	42
2364	invalid anonymous union - member function is not allowed.....	43
2365	anonymous union at global or namespace scope must be declared static.....	43
2375	declaration requires a typedef name.....	44
2381	extra ";" ignored.....	44
2487	inline %n cannot be explicitly instantiated.....	44
2513	a value of type %t1 cannot be assigned to an entity of type %t2.....	44
2546	transfer of control bypasses initialization of: "variable".....	45
2549	"variable" is used before its value is set.....	45
2550	%n was set but never used.....	45
2656	transfer of control into a try block.....	46
2767	conversion from pointer to smaller integer.....	46
2815	type qualifier on return type is meaningless.....	46
2826	%n was never referenced.....	46
2830	%n has no corresponding operator delete%s (to be called if an exception is thrown during initialization of an allocated object).....	47
2836	returning reference to local variable.....	47
2837	omission of explicit type is nonstandard ("int" assumed).....	47
2940	missing return statement at end of non-void function "function".....	48
2951	return type of function "main" must be int.....	48
2991	extra braces are nonstandard.....	48
3000	a storage class may not be specified here.....	48
3051	standard requires that %n be given a type by a subsequent declaration (\ "int\" assumed).....	48

3055 types cannot be declared in anonymous unions.....	49
3056 returning pointer to local variable.....	49
3105 #warning directive: %s.....	49
3138 format argument does not have string type.....	50
3145 %t1 would have been promoted to %t2 when passed through the ellipsis parameter; use the latter type instead.....	50
3197 the prototype declaration of %nfd is ignored after this unprototyped redeclaration.....	50
3290 Passing a non-POD object to a function with variable arguments has undefined behavior. Object will be copied onto the stack instead of using a constructor.....	51
3348 declaration hides %nd.....	51
3353 %n has no corresponding member operator delete%s (to be called if an exception is thrown during initialization of an allocated object).....	51
3750 "\" followed by white space is not a line splice.....	52
4212 mismatch between character pointer types %t1 and %t2.....	52
4225 suggest parentheses around comparison in operand of %sq.....	52
4227 padding struct with %s1 bytes to align member %sq2.....	53
4228 64 bit migration: conversion from %t1 to a more strictly aligned type %t2 may cause misaligned access	53
4229 64 bit migration: conversion from %t1 to %t2 may truncate value.....	53
4230 64 bit migration: conversion from %t1 to %t2 may cause target of pointers to have a different size.....	54
4231 64 bit migration: conversion between types of different sizes has occurred (from %t1 to %t2).....	54
4232 conversion from %t1 to a more strictly aligned type %t2 may cause misaligned access.....	54
4235 conversion from %t1 to %t2 may lose significant bits.....	55
4237 type cast from %t1 to %t2 may cause sign extension to a larger size integer.	55
4239 case type mismatch with switch expression type	55
4241 redeclaration, function %nod was previously called without a prototype.....	56
4242 No prototype or definition in scope for call to %sq.....	56
4243 function declared with empty parentheses, consider replacing with a prototype.....	56
4244 extern storage class used with a function definition.....	56
4245 storage class used with a data definition.....	57
4247 function called with different argument counts (%s vs. %s2).....	57
4248 comparison of unsigned integer with a signed integer.....	57
4249 64 bit migration: value could be truncated before cast to bigger sized type.....	57
4251 the assignment has an excessive size for a bit field.....	58
4253 unsigned value cannot be less than zero.....	58
4255 padding size of struct %sq1 with %s2 bytes to alignment boundary.....	58
4259 suggest parentheses around the operands of %sq.....	59
4264 padding size of struct anonymous with %s bytes to alignment boundary.....	59
4272 conversion from %t1 to %t2 may lose sign.....	60
4273 floating-point equality and inequality comparisons may be inappropriate due to round off common in floating-point computation.....	60
4274 comparison of pointer with integer zero.....	60
4275 constant out of range (%s) for the operator.....	60
4276 relational operator %sq always evaluates to 'false'.....	61
4277 logical AND with a constant, do you mean to use '&'?.....	61
4278 the sub expression in logical expression is a constant.....	61
4279 the expression depends on order of evaluation.....	62
4281 assignment in control condition.....	62
4286 return non-const handle to non-public data member may undermine encapsulation.....	62
4289 endian porting: the definition of the union may be endian dependent.....	63
4290 endian porting: the initialization of char array may be endian dependent.....	63
4292 endian porting: the dereference of cast pointer may be endian dependent.....	63

4295 abstract function type declared with empty parentheses, consider replacing with parameter list or void.	64
4298 64 bit migration: addition result could be truncated before cast to bigger sized type.....	64
4299 64 bit migration: multiply result could be truncated before cast to bigger sized type.....	64
4300 Overflow while computing constant in left shift operation.....	65
4301 expression has no effect.....	65
4314 if statement without body, did you insert an extra ';'.....	65
4315 %s loop without body, did you insert an extra ';'.....	65
4354 One of the operands of the %sq operation is a string literal, strcmp() is recommended for such comparison.....	66
4355 the initializer for %n is greater than %s.....	66
4356 operand of sizeof is a constant rvalue, this might not be what you intended.....	66
4357 octal escape sequence "%s" is followed by decimal character '%s2'.....	66
4360 size of types of consecutive bitfields are different, bitfield packing behavior may be different across compiler.....	67
4361 64 bit migration: size of types of consecutive bitfields are different, bitfield packing behavior may be different across compilers.....	67
4363 possible mismatch in parameters passed to %n, previously seen %p.....	68
4364 endian porting: type cast is endian dependent.....	68
4365 endian porting: the definition of the union may be endian dependent.....	68
4370 Control flows into the switch case from the previous case value.....	69
4372 Potential overflow in arithmetic expression involving time_t/clock_t values.....	69
4373 non arithmetic integer conversion resulted in a change of sign.....	70
20035 variable %s is used before its value is set.....	70
20036 variable %s (field %s) is used before its value is set.....	70
20037 variable %s may be used before its value is set.....	71
20048 %s "%s" has incompatible type with previous declaration at line %s in file "%s".....	71
20072 variable %s is partially uninitialized when used.....	72
20073 variable %s may be partially uninitialized when used.....	72
20074 variable %s (field "%s") may be used before its value is set.....	72
20200 Potential null pointer dereference %s%s is detected %s.....	73
20201 Memory leak is detected.....	73
20202 Allocated memory may potentially be leaked %s.....	74
20203 Potential out of scope use of local %s %s.....	74
20206 Out of bound access (%s).....	75
20208 Forming out of bound address(%s).....	75
20210 Mismatch in allocation and deallocation.....	76
20213 Potential write to read only memory %s%s is detected.....	76
20229 variable "%s" is uninitialized if the loop %s is not executed.....	77
20231 variable "%s" (field "%s") may be used before its value is set if the loop %s is not executed....	77

About This Document

This document provides a brief description for a partial list of diagnostics emitted by Cadvise. There are four sections for each description as mentioned below:

- **The Cause section**
Discusses the possible reasons for the diagnostic to be emitted.
- **The Example section**
Provides relevant sample code segments that result in the specific diagnostic. The sample code segments are not complete, compilable programs - you might have to include the necessary header files, provide the code segment within main() function (if one is not provided in the code segment) and use necessary compiler options (such as +w, +wlint) to reproduce the diagnostic.
- **The Action section**
Suggests possible solutions to avoid the diagnostic.
- **The Reference section**
Consists of relevant section numbers from the standard document if the diagnostic pertains to any language standard.

Intended Audience

This document is intended for HP Code Advisor users who resolve issues through the diagnostics emitted by HP aC++/HP C.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to:

cadvise-help@lists.hp.com

1 Diagnostics Categorization

This chapter addresses the diagnostics categorization for HP Code Advisor.

NOTE:

- Some of the diagnostic numbers given below do not have a corresponding diagnostic description. In case you need a diagnostic description for a diagnostic number, please send an email to:
cadvice-help@lists.hp.com
- The diagnostics with number greater than 20000 are more effective when cross-file analysis is enabled. For more information on cross-file analysis, see *HP Code Advisor User's Guide*.

Default Diagnostics

Following table lists all the diagnostic messages that are emitted even when you do not provide any command line diagnostic options.

Table 1 Default diagnostics

Diagnostic number	Diagnostic message
2021	type qualifiers are meaningless in this declaration
2031	expression must have integral type
2037	the #endif for this directive is missing
2042	operand types are incompatible (%t1 and %t2)
2056	operand of sizeof may not be a function
2069	integer conversion resulted in truncation
2070	incomplete type is not allowed
2080	a storage class may not be specified here
2081	more than one storage class may not be specified
2085	invalid storage class for a parameter
2086	invalid storage class for a function
2094	the size of an array must be greater than zero
2099	a declaration here must declare a parameter
2101	%sq has already been declared in the current scope
2107	zero-length bit field must be unnamed
2108	signed bit field of length 1
2137	expression must be a modifiable lvalue
2138	taking the address of a register variable is not allowed
2139	taking the address of a bit field is not allowed
2140	too many arguments in function call
2144	a value of type %t1 cannot be used to initialize an entity of type %t2
2147	declaration is incompatible with %nfd

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
2149	a global-scope declaration may not have this storage class
2155	old-fashioned assignment operator
2157	expression must be an integral constant expression
2161	unrecognized #pragma
2165	too few arguments in function call
2167	argument of type %t1 is incompatible with parameter of type %t2
2174	expression has no effect
2175	subscript out of range
2178	\ "&" applied to an array has no effect
2181	argument is incompatible with corresponding format string conversion
2186	pointless comparison of unsigned integer with zero
2187	use of \ "=" where \ "==" may have been intended
2188	enumerated type mixed with another type
2224	the format string requires additional arguments
2225	the format string ends before this argument
2226	invalid format string conversion
2231	declaration is not visible outside of function
2236	controlling expression is constant
2247	%n has already been defined
2253	expected a \ ",\"
2277	invalid friend declaration
2284	NULL reference is not allowed
2300	a pointer to a bound function may only be used to call the function
2302	%n has already been defined
2313	type qualifier is not allowed on this function
2315	the object has cv-qualifiers that are not compatible with the member function
2335	linkage specification is not allowed
2340	value copied to temporary, reference to temporary used
2341	\ "operator%s\" must be a member function
2347	default argument is not allowed
2368	%n defines no constructor to initialize the following:
2370	%n has an uninitialized const field
2414	delete of pointer to incomplete class
2430	returning reference to local temporary
2433	qualifiers dropped in binding reference of type %t1 to initializer of type %t2

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
2460	declaration of %sq hides function parameter
2472	member function typedef (allowed for cfront compatibility)
2490	%n cannot be instantiated -- it has been explicitly specialized
2495	global %n1 used instead of %n2 (cfront compatibility)
2497	declaration of %sq hides template parameter
2514	pointless comparison of unsigned integer with a negative constant
2524	non-const function called for const object (anachronism)
2533	handler is potentially masked by previous handler for type %t
2540	support for exception handling is disabled
2549	%n is used before its value is set
2552	exception specification is not allowed
2553	external/internal linkage conflict for %nfd
2554	%nf will not be called for implicit or explicit conversions
2609	this kind of pragma may not be used here
2611	overloaded virtual function %no1 is only partially overridden in %n2
2617	pointer-to-member-function cast to pointer to function
2640	very large entity in program prevents generation of precompiled header file
2643	\ <code>"restrict"</code> is not allowed
2645	%sq is an unrecognized <code>__declspec</code> attribute
2650	calling convention specified here is ignored
2654	declaration modifiers are incompatible with previous declaration
2655	the modifier %sq is not allowed on this declaration
2656	transfer of control into a try block
2657	inline specification is incompatible with previous %nod
2662	call of pure virtual function
2708	incrementing a bool value is deprecated
2715	based does not precede a pointer operator, <code>__based</code> ignored
2720	redeclaration of %n is not allowed to alter its access
2780	reference is to %nd1 -- under old for-init scoping rules it would have been %nd2
2783	empty comment interpreted as token-pasting operator <code>"##"</code>
2784	a storage class is not allowed in a friend declaration
2793	explicit specialization of %n must precede its first use
2825	virtual inline %n was never defined
2829	\ <code>"double"</code> used for <code>"long double"</code> in generated C code
2830	%n has no corresponding operator <code>delete%</code> s (to be called if an exception is thrown during initialization of an allocated object)

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
2836	returning reference to local variable
2855	return type is not identical to return type %t of overridden virtual function %n
2867	declaration of \"size_t\" does not match the expected type %t
2902	type qualifier ignored
2925	type qualifiers on function types are ignored
2959	declared size for bit field is larger than the size of the bit field type; ; truncated to %s bits
2991	extra braces are nonstandard
2997	%np2 is hidden by %no1 - virtual function override intended?
3000	a storage class may not be specified here
3046	floating-point value cannot be represented exactly
3050	imaginary *= imaginary sets the left-hand operand to zero
3091	invalid attribute for %t
3092	invalid attribute for %n
3093	invalid attribute for parameter
3097	attribute %sq ignored
3100	the \"packed\" attribute is ignored in a typedef
3108	the \"transparent_union\" attribute is ignored on incomplete types
3109	%t cannot be transparent because %n does not have the same size as the union
3110	%t cannot be transparent because it has a field of type %t which is not the same size as the union
3117	an asm name is ignored in a typedef
3119	modifier letter '%s' ignored in asm operand
3129	register \"%s\" clobbered more than once
3135	\"format\" attribute applied to %n which does not have variable arguments
3136	first substitution argument is not the first variable argument
3137	format argument index is greater than number of parameters
3138	format argument does not have string type
3142	attribute does not apply to non-function type %t
3143	arithmetic on pointer to void or function type
3156	nonstandard cast on lvalue ignored
3159	the auto specifier is ignored here (invalid in standard C/C++)
3160	a reduction in alignment without the \"packed\" attribute is ignored
3168	an asm name is ignored on a non-register automatic variable
3169	inline function also declared as an alias; definition ignored
3185	argument of upc_blocksizeof is a pointer to a shared type (not shared type itself)
3187	branching into or out of a upc_forall loop is not allowed

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
3197	the prototype declaration of %nfd is ignored after this unprototyped redeclaration
3202	call of zero constant ignored
3210	declaration of %sq hides handler parameter
3211	nonstandard cast to array type ignored
3213	field uses tail padding of a base class
3214	GNU C++ compilers may use bit field padding
3215	%nd was declared <code>\\"deprected\\"</code>
3217	unrecognized format function type %sq ignored
3218	base class %no1 uses tail padding of base class %no2
3220	requested initialization priority is reserved for internal use
3221	this anonymous union/struct field is hidden by %nd
3222	invalid error number
3223	invalid error tag
3224	expected an error number or error tag
3225	size of class is affected by tail padding
3235	nonstandard conversion between pointer to function and pointer to data
3262	earlier <code>__declspec(align(...))</code> ignored
3272	a throw expression may not have pointer-to-incomplete type
3283	GNU layout bug not emulated because it places virtual base %no1 outside %no2 object boundaries
3284	virtual base %no1 placed outside %no2 object boundaries
3286	reduction in alignment ignored
3287	const qualifier ignored
3290	Passing a non-POD object to a function with variable arguments has undefined behavior. Object will be copied onto the stack instead of using a constructor.
3291	a non-POD class type cannot be fetched by <code>va_arg</code>
3294	integer operand may cause fixed-point overflow
3296	fixed-point value cannot be represented exactly
3301	%nf declares a non-template function -- add <code><></code> to refer to a template instance
3302	operation may cause fixed-point overflow
3305	function declared with <code>\\"noreturn\\"</code> does return
3306	asm name ignored because it conflicts with a previous declaration
3307	class member typedef may not be redeclared
3308	taking the address of a temporary
3309	attributes are ignored on a class declaration that is not also a definition
3310	fixed-point value implicitly converted to floating-point type
3342	<code>const_cast</code> to enum type is nonstandard

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
3346	function returns incomplete class type %t
3347	%n has already been initialized; the out-of-class initializer will be ignored
3359	variable attributes appearing after a parenthesized initializer are ignored
3360	the result of this cast cannot be used as an lvalue
3361	negation of an unsigned fixed-point value
3377	friend specifier is not allowed in a class definition; friend specifier is ignored
3386	storage specifier ignored
3387	dllexport and dllimport are ignored on class templates
3388	base class dllexport/dllimport specification differs from that of the derived class
3394	field of class type without a DLL interface used in a class with a DLL interface
3399	nonstandard reinterpret_cast
3400	positional format specifier cannot be zero
3427	offsetof applied to non-POD types is nonstandard
3433	no prior push_macro for %sq
3443	__real/__imag applied to real value
3444	%nd was declared \"deprecated (%sq)\"
3446	dllimport/dllexport applied to a member of an unnamed namespace
3546	argument must be a constant null pointer value
3547	insufficient number of arguments for sentinel value
3548	sentinel argument must correspond to an ellipsis parameter
3550	#pragma start_map_region already active: pragma ignored
3552	%n cannot be used to name a destructor (a type name is required)
3565	anonymous union qualifier is nonstandard
3566	anonymous union qualifier is ignored
3568	__declspec(%s) ignored (it has no meaning for a C struct)
3570	nonstandard specifier ignored
3571	attributes are ignored on an enum declaration that is not also a definition
3573	a condition declaration for an array is always true
3575	visibility attribute ignored because it conflicts with a previous declaration
3580	declaration hides built-in %n
3581	declaration overloads built-in %n
3609	asm name ignored for previously defined entity
3621	no parameter has pointer type
3622	null argument provided for parameter marked with attribute \"nonnull\"
3625	routine is both \"inline\" and \"noinline\" (\"noinline\" assumed)

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
3627	attribute <code>"cleanup"</code> requires automatic storage duration
3628	attribute <code>"cleanup"</code> does not apply to parameters
3644	definition at end of file not followed by a semicolon or a declarator
3647	extra arguments ignored
3650	result of call is not used
3651	attribute is ignored for void return type
3652	attribute requires pointer-to-function type
3653	<code>dllimport/dllexport</code> is ignored on redeclaration using a qualified name
"3750" (page 52)	<code>"\"</code> followed by white space is not a line splice
4006	this pragma allowed only before any statements or declarations in a block scope, ignored
4007	this pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope
4045	non-constant initialization performed at runtime
4046	extra text after expected end of preprocessing directive, extra text ignored
4048	pragma expects either <code>OFF_IF_NO_KEY_BEGIN</code> or <code>OFF_IF_NO_KEY_END</code> , pragma ignored" <code>ec_pragma_vtable_bad_arg</code>
4052	this pragma must be in the outermost block of a loop, pragma ignored
4053	this pragma must be in the outermost block of a loop or an if/else statement, pragma ignored
4058	argument to <code>#pragma align</code> must be 0 or an integral power of two less than or equal to <code>%s</code> , pragma ignored
4059	<code>#pragma align</code> must immediately precede the declaration of a static or file scope variable, pragma ignored
4060	<code>%n</code> is undefined in this translation unit, <code>#pragma align %p</code> ignored
4062	specified alignment is smaller than that required by the type of <code>%n</code> , pragma ignored
4071	argument to <code>#pragma unalign</code> must a power of two
4072	<code>#pragma unalign</code> must immediately precede the declaration of a typedef, pragma ignored
4073	specified alignment is greater than the existing alignment of the type <code>%t</code> , pragma ignored
4094	left operand of <code>"->"</code> or <code>".\"</code> operator will be evaluated, though right operand is a static data member
4095	left operand of <code>"->"</code> or <code>".\"</code> operator will be evaluated, though right operand is a static member function
4120	the shared variable <code>%sq</code> will be updated by all the threads in the parallel region
4136	the first expression of OpenMP for statement, is not a simple integral assignment expression.
4137	OpenMP loop control variable is not a signed integer type.
4138	the third expression of OpenMP for statement does not meet the requirements of a parallel loop.
4139	the second expression of OpenMP for statement does not meet the requirements of a parallel loop.
4157	in this statement, the variable <code>%sq</code> is not declared within the scope of a parallel region with a <code>default(none)</code> clause. The variable is assumed shared.

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
4162	This feature, within an OpenMP directive region, is not supported by this version of C++. All surrounding OpenMP processing pragmas are being ignored.
4173	the maximum alignment of local variables is 16 bytes
4174	over alignment of local array %n is not possible
4175	#pragma align %p supercedes other alignments of %n
4176	redeclaration of enumerator %s %p
4189	a non-POD class type used in offsetof macro
4191	the maximum alignment associable with the type %t is %s bytes
4192	%t unaligned with #pragma unalign %p which supercedes this alignment
4193	%t is over aligned, over alignment of local arrays is not possible
4194	%n is undefined in this translation unit, alignment ignored
4195	Virtual operator= function found, behavior may be different than previous compiler, consider creating an operator= function
4196	lexicographically decreasing version_id not allowed
4203	The assertion was not true, %sq
4204	The implicit conversion from const to nonconst qualification for string literals is deprecated
4208	const variable used in C constant context
4209	%no is ambiguous with another template in an available namespace
4213	found a use of a class type containing a pointer to member which is a POD in aCC version 6 and not a POD type in version 5. This incompatibility may cause failures in applications built with objects from both releases. For aCC version 5 compatibility use
4219	64 bit migration: type conversion may truncate value
4220	64 bit migration: conversion to a more strictly aligned type may cause misaligned access
4221	64 bit migration: type conversion may cause target of pointers to have a different size
4222	64 bit migration: conversion between types of different sizes has occurred
4225	suggest parentheses around comparison in operand of %sq
4234	type conversion may lose significant bits
4236	type cast may cause sign extension to a larger size integer
4240	declaration of "\"ptrdiff_t\"" does not match the expected type %t
4256	taking the address of a data member is not allowed
4258	Union type containing copy constructor is incompatible with aC++ version A.05.xx when passed by value. Use -WC,-nonstdunion,on have compatibility with aC++ version A.05.xx
4259	suggest parentheses around the operands of %sq
4260	pragma does not precede a loop, pragma ignored
4261	initialization on operand of type __fprog performed at runtime
4266	conversion from unaligned field %sq to %t may cause misaligned access
4271	type conversion may lose sign

Table 1 Default diagnostics *(continued)*

Diagnostic number	Diagnostic message
4288	Object %sq in pragma noptrs_to_globals may not have its address taken, pragma ignored
4294	the use of pragma %s is not allowed in this environment
4301	expression has no effect
4308	case label constant out of range (%s) for switch expression
4313	no prototype or definition in scope for call to memory allocation routine %sq
4321	One or more source files were inaccessible (error: %s) while emitting the source context in diagnostics, emitting column number instead.
4322	The asm declaration is ignored.
4343	#pragma diag_pop' attempted with an empty diagnostic pragma stack
4345	tls_model attribute ignored because it conflicts with a previous declaration
4346	tls_model local-exec attribute requires -exec option
4355	the initializer for %n is greater than %s
4356	operand of sizeof is a constant rvalue, this might not be what you intended.

+wall Diagnostics

Following table lists all the diagnostic messages that cadvice emits when you use the `+wall` command line option.

Table 2 +wall diagnostics

Column Head	Column Head
20229	variable "%s" is uninitialized if the loop %s is not executed
20230	variable "%s" may be partially uninitialized if the loop %s is not executed
20231	variable "%s" (field "%s") may be used before its value is set if the loop %s is not executed

+wlint Diagnostics

Following table lists all the diagnostic messages that cadvice emits when you use the `+wlint` command line option.

Table 3 +wlint diagnostics

Diagnostic number	Diagnostic message
"2180" (page 31)	argument is incompatible with formal parameter
2193	zero used for undefined preprocessing identifier %sq
2228	trailing comma is nonstandard
2260	explicit type is missing ("int" assumed)
2267	old-style parameter list (anachronism)
2324	duplicate friend declaration
2381	extra ";" ignored
2399	%n has an operator new% <i>s</i> () but no default operator delete% <i>s</i> ()
2401	destructor for base class %nod is not virtual

Table 3 +wlint diagnostics (continued)

Diagnostic number	Diagnostic message
2550	%n was set but never used
2767	conversion from pointer to smaller integer
2108	signed bit field of length 1
2815	type qualifier on return type is meaningless
2940	missing return statement at end of non-void
3051	standard requires that %n be given a type by a subsequent declaration ("int" assumed)
3053	conversion from integer to smaller pointer
3348	declaration hides %nd
4324	declaration hides %nd
3373	implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
4224	conversion from explicitly unaligned type may cause misaligned access
4233	conversion from explicitly unaligned type %t1 to %t2 may cause misaligned access
4239	case type mismatch with switch expression type
4240	declaration of "ptrdiff_t" does not match the expected type %t
4241	redeclaration, function %nod was previously called without a prototype
4242	no prototype or definition in scope for call to %sq
4243	function declared with empty parentheses, consider replacing with a prototype
4247	function called with different argument counts (%s vs. %s2)
4249	64 bit migration: value could be truncated before cast to bigger sized type
4250	the length of boolean type bit field is greater than 1
4251	the assignment has an excessive size for a bit field
4252	the assignment may have an excessive size for a bit field
4253	unsigned value cannot be less than zero
4274	comparison of pointer with integer zero
4279	the expression depends on order of evaluation
4285	operator= does not have a check for the source and destination addresses being non-identical
4295	abstract function type declared with empty parentheses, consider replacing with parameter list or void.
4296	%s operation on boolean type
4297	boolean value is used as array index
4299	64 bit migration: multiply result could be truncated before cast to bigger sized type
4300	Overflow while computing constant in left shift operation
4313	no prototype or definition in scope for call to memory allocation routine %sq
4314	if statement without body, did you insert an extra ';'?
4354	One of the operands of the %sq operation is a string literal, strcmp() is recommended for such comparison

Table 3 +wlint diagnostics (continued)

Diagnostic number	Diagnostic message
4357	octal escape sequence "%s" is followed by decimal character '%s2'
20200	Potential null pointer dereference %s%s is detected %s
20201	Memory leak is detected
20202	Allocated memory may potentially be leaked %s
20203	Potential out of scope use of local %s %s
20204	Potential out of scope use of alloca return address %s %s
20205	Pointer is used after free
20206	Out of bound access (%s)
20207	Possible out of bound access (%s)
20208	Forming out of bound address (%s)
20209	Forming possible out of bound address (%s)
20210	Mismatch in allocation and deallocation
20211	Use of potential null reference %s is detected %s
20213	Potential write to readonly memory %s%s is detected
20226	Found Type(s) thrown from function are uncaught: %s
20227	Dereference pointer through a non-pointer object: '%s'
20228	Signal handler '%s' calls function '%s' which may not be Async Signal Safe
20250	Indirect callsite has void return which mismatches the target function '%s''s signature

+w64bit Diagnostics

Following table lists all the diagnostic messages that cadvice emits when you use the +w64bit command line option.

Table 4 +w64bit diagnostics

Diagnostic number	Diagnostic message
2767	implicit typename option can be used only when compiling C++
3373	implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
3374	explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
3375	conversion from pointer to same-sized integral type (potential portability problem)
4219	64 bit migration: type conversion may truncate value
4220	64 bit migration: conversion to a more strictly aligned type may cause misaligned access
4221	64 bit migration: type conversion may cause target of pointers to have a different size
4222	64 bit migration: conversion between types of different sizes has occurred
4228	64 bit migration: conversion from %t1 to a more strictly aligned type %t2 may cause misaligned access
4229	64 bit migration: conversion from %t1 to %t2 may truncate value

Table 4 +w64bit diagnostics (continued)

Diagnostic number	Diagnostic message
4230	64 bit migration: conversion from %t1 to %t2 may cause target of pointers to have a different size.
4231	64 bit migration: conversion between types of different sizes has occurred (from %t1 to %t2)
4249	64 bit migration: value could be truncated before cast to bigger sized type.
4280	64 bit migration: comparison of unsigned integer with signed long
4298	64 bit migration: addition result could be truncated before cast to bigger sized type
4299	multiply result could be truncated before cast to bigger sized type
4313	no prototype or definition in scope for call to memory allocation routine %sq

+wendian Diagnostics

Following table lists all the diagnostic messages that cadvise emits when you use the `+wendian` command line option.

Table 5 +wendian diagnostics

Diagnostic number	Diagnostic message
4289	endian porting: the definition of the union may be endian dependent
4290	endian porting: the initialization of char array may be endian dependent
4291	endian porting: the read/write of the buffer may be endian dependent
4292	endian porting: the dereference of cast pointer may be endian dependent
4364	endian porting: type cast is endian dependent

+wsecurity Diagnostics

Following table lists all the diagnostic messages that cadvise emits when you use the `+wsecurity` command line option.

Table 6 +wsecurity diagnostics

Diagnostic number	Diagnostic message
20111	(SECURITY) Tainted data may be used in data length computation%s
20112	(SECURITY) Tainted data may be copied to the target buffer%s
20113	(SECURITY) Data buffer may contain tainted data
20114	(SECURITY) Tainted value may be used in loop exit condition computation%s
20115	(SECURITY) Tainted value may be used as format string%s
20116	(SECURITY) Tainted value may be used as path or file name%s
20117	(SECURITY) Tainted value may be used in array index expression%s
20118	Tainted value may be used in pointer arithmetic expression%s
20119	(SECURITY) Unsafe API '%s' is used.%s
20132	(SECURITY) %%s is used in the format string for input, potential buffer overflow may occur in %s argument
20212	(SECURITY) Improper creation of temporary file using 'fopen' can lead to race condition. Use 'open' with O_CREAT O_EXCL flag instead

+wlock Diagnostics

Following table lists diagnostic messages that cadvice emits when you use the `+wlock` command line option.

Table 7 +wlock diagnostics

Diagnostic number	Diagnostic message
20220	Trying to lock an already held lock
20221	Trying to unlock a lock not held
20222	Failed to release lock %s is detected at program exit
20223	Trying to unlock a lock held conditionally
20224	Potential access of global variable %s without any locks held
20225	Potential access of global variable %s without appropriate lock held

+wperfadvice Diagnostics

Following table lists all the diagnostic messages that cadvice emits when you use the `+wperfadvice` command line option.

Table 8 +wperfadvice diagnostics

Diagnostic number	Diagnostic message
4319	performance advice: passing a large (%s1 byte) parameter by value is inefficient, consider passing %s2
4320	performance advice: <code>std::endl</code> is expensive because it flushes the stream. If you do not want to flush the stream replace ' <code>std::endl</code> ' with '\\\\n' or define macro <code>_HP_NONSTD_FAST_Iostream</code>
11042	performance advice: Exceeding memory limits when compiling procedure %1\$. Consider splitting it in smaller procedures.
11043	performance advice: Loop is not pipelined because of resource constraints. Consider splitting the loop.
11044	performance advice: Loop is not pipelined because of recurrence constraints. Consider restructuring the loop to avoid dependencies.
20300	performance advice: Minimal optimizations are performed at +O%. Consider using an optimization level ≥ 2 for better performance
20301	performance advice: Optimization is turned off for the routine '%s'. If '#pragma optimize off' is used, consider removing it
20302	performance advice: For better performance, consider using profile-based optimization.
20310	performance advice: Routine '%s' in '%s' is indirectly called %s times (%s% of total calls). Conditionally promoting it to a direct call could reduce indirect call overhead and enable additional optimizations.
20311	performance advice: Loop iterated %s times during %s% of total invocations. Multi-versioning the loop with the constant iteration count could enable a dditional optimizations. If this loop is one of the inner loops of a loop nest, consider multi-versioning it along with the enclosing loops.
20312	performance advice: Parallelizing this loop will likely provide better performance. Consider using <code>+Oautopar</code> to do this automatically.
20313	performance advice: Multi-versioning this loop and parallelizing the higher trip-count version will likely provide better performance. Consider using <code>+ Oautopar</code> to do this automatically.

Table 8 +wperfadvise diagnostics (continued)

Diagnostic number	Diagnostic message
20314	performance advice: The data being accessed is less strictly aligned than the type of the address. Certain optimizations may be prevented.
20315	performance advice: Frequently called routine '%s' cannot be inlined because it is not defined in the current load module.
20316	performance advice: Parallelizing this loop would require %s. Consider using +Oautopar to do this automatically.

+w/+w1 Diagnostics

Cadvise emits all the diagnostics in the above mentioned categories and a few additional diagnostics when you use the +w/+w1 option. Following table lists the additional diagnostic messages that cadvise emits when you use the +w/+w1 command line option.

Table 9 +w/+w1 diagnostics

Diagnostic number	Diagnostic message
2108	signed bit field of length 1
2174	expression has no effect
2193	zero used for undefined preprocessing identifier %sq
2261	access control not specified (%sq by default)
2399	%n has an operator new%s() but no default operator delete%s()
2400	%n has a default operator delete%s() but no operator new%s()
2401	destructor for base class %nod is not virtual
2451	omission of %sq is nonstandard
2479	%n redeclared "\"inline\" after being called
2534	use of a local type to specify an exception
2535	redundant type in exception specification
2652	calling convention is ignored for this type
2679	%n cannot be inlined
2863	effect of this "\"#pragma pack\" directive is local to %n
2866	ec_exception_specification_ignored
3051	standard requires that %n be given a type by a subsequent declaration (\""int\" assumed)
3189	affinity has shared type (not pointer to shared)
3257	potentially narrowing conversion when compiled in an environment where int, long, or pointer types are 64 bits wide
3348	declaration hides %nd
3353	%n has no corresponding member operator delete%s (to be called if an exception is thrown during initialization of an allocated object)
4049	vtable generation for classes with no key functions already %s, pragma ignored
4080	explicit request to instantiate %n ignored because of directive %p
4161	OpenMP directives are not active in this compilation because the +Oopenmp option was not specified on the command line. All OpenMP processing pragmas are being ignored.

Table 9 +w/+w1 diagnostics (continued)

Diagnostic number	Diagnostic message
4163	Conversion of string literal from const to non-const has been deprecated
4217	comparison of enum (represented as an unsigned integer) with zero
4218	comparison of unsigned integer with an enumerated value of zero
4227	padding struct with %s1 bytes to align member %sq2
4241	redeclaration, function %nod was previously called without a prototype
4242	no prototype or definition in scope for call to %sq
4243	function declared with empty parentheses, consider replacing with a prototype
4246	ignoring return value of %n
4247	function called with different argument counts (%s vs. %s2)
4248	comparison of unsigned integer with a signed integer
4250	the length of boolean type bit field is greater than 1
4253	ec_unsigned_cannot_less_than_zero
4253	padding size of struct %sq1 with %s2 bytes to alignment boundary
4263	padding struct with %s bytes to align member <anonymous>
4264	padding size of struct <anonymous> with %s bytes to alignment boundary
4280	ec_lp64_unsigned_compare_with_signed
4281	assignment in control condition
4295	abstract function type declared with empty parentheses, consider replacing with parameter list or void
4296	%s operation on boolean type
4297	boolean value is used as array index
4301	expression has no effect
4324	declaration hides %nd
4356	operand of sizeof is a constant rvalue, this might not be what you intended

Diagnostic messages turned off by default

Cadvice does not emit certain diagnostics under any command line options. To get these diagnostics you must explicitly mention the `+Ww` command line option followed by the `<diag no>`. Following table lists the diagnostic messages that are turned off by default.

Table 10 Diagnostic messages that are turned off by default

Diagnostic number	Diagnostic message
4246	ignoring return value of %n
4325	64 bit migration: struct %sq (recursively) contains a long or pointer field, its size will change based on the data model
4360	size of types of consecutive bitfields are different, bitfield packing behavior may be different across compiler
4361	64 bit migration: size of types of consecutive bitfields are different, bitfield packing behavior may be different across compilers

Table 10 Diagnostic messages that are turned off by default *(continued)*

Diagnostic number	Diagnostic message
4363	possible mismatch in parameters passed to %n, previously seen %p
4365	endian porting: the definition of the union may be endian dependent
4370	control flows into the switch case from the previous case value
4372	potential overflow in arithmetic expression involving time_t/clock_t values
4373	non arithmetic integer conversion resulted in a change of sign

2 Diagnostics Details

This chapter provides the detailed description for a subset of diagnostics emitted by HP Code Advisor.

2009 nested comment is not allowed

Cause:

Opening comment delimiter, `/*`, found inside delimited comment. A previous comment may be missing its closing delimiter. C comments delimited by `/* */` do not nest.

Example:

```
/* comment1          // missing */ for comment1
/* comment2          // warning 2009 here
*/
```

Action:

Check if a previous comment is missing its terminating `*/` or if you are attempting to "comment out" a section of code that contains a delimited comment.

Reference:

C99 6.4.9, ANSI/ISO C++ 2.7(1)

2028 expression must have a constant value

Cause:

In many cases, initializations need to be constant expressions. This diagnostic is emitted if the initialization expression is not a constant expression.

Example:

```
// error in C-mode
void foo() {
    int i = 10;
    static int j = i;
}
```

Action:

Provide constant expressions for initialization.

Reference:

C99 6.6

2042 operand types are incompatible (%t1 and %t2)

Cause:

Operation might be invalid due to incompatible operand types.

Example:

```
#define DT 10
typedef char* caddr;
int main() {
    caddr data = "0x1";
    if (data == DT)
        return 1;
}
```

Action:

Use valid/compatible operand types or use a cast operator to convert one operand type to the other type.

Reference:

2047 incompatible redefinition of macro %nod

Cause:

A #define preprocessing directive has redefined a macro whose previous definition contained an error or warning. Normally, the compiler will issue a warning if a macro is redefined to something other than the previous definition. However, if the previous definition caused a warning or error to be generated, this informational message is output instead.

Example:

```
#define KM_SLEEP          0
#define KM_SLEEP          1
```

Action:

Do not redefine a macro without first undefining it.

Reference:

C99 6.9.3

2054 Too few arguments in macro invocation

Cause:

Some parameter/s is/are omitted during macro invocation.

Example:

```
#define FOO(a, b, c) (( a << 3 ) | b)
int a, b;
int main() {
    (void) FOO(a, b);
}
```

Action:

Pass right number of parameters during macro invocation.

Reference:

2063 shift count is too large

Cause:

The compiler has detected a shift count that is greater than or equal to the size of the operand to be shifted. This may not be your intent, as the result contains none of the original bits of the operand.

Example:

```
int main() {
    int a = 1 << 32;
}
```

Action:

Avoid using the shift count greater than number of bits a given type can hold.

Reference:

2064 declaration does not declare anything

Cause:

The declaration does not declare anything. The C standard requires that a declaration must declare at least a tag, an enumeration constant or a declarator.

Example:

```
int;
```

Action:

Correct or remove the declaration.

Reference:

C99 6.7.2, ANSI/ISO C++ 6.7(2)

2068 integer conversion resulted in a change of sign

Cause:

Conversion of integer resulted in sign change. Either an unsigned 1-bit bitfield was assigned -1, or a signed 1-bit bitfield was assigned 1.

Example:

```
#define NEGATIVE (-1)
int main() {
    unsigned int a = NEGATIVE;
}
```

Action:

Conversion from a signed int to unsigned leads to change of sign. This might result in an unexpected behavior of the code. If the sign of the integer is not a concern, then this diagnostic can be ignored. Otherwise, do not assign a signed value to unsigned type and vice versa.

Reference:

2069 integer conversion resulted in truncation

Cause:

Conversion of integer resulted in truncation. This causes loss of some digits.

Example:

```
#define FCD_NEGATIVE 390000000
int main() {
    short int a = FCD_NEGATIVE;
}
```

Action:

Conversion from a larger integral type to a smaller integral type leads to truncation. This might result in an unexpected behavior of the code. Hence, avoid doing such conversions.

Reference:

2077 this declaration has no storage class or type specifier

Cause:

Not a conversion function, constructor, or destructor, therefore something where a type specifier is expected.

Example:

```
struct X {  
    i;  
};
```

Action:

Provide a declaration specifier.

Reference:

C99 6.7.2

2082 storage class is not first

Cause:

The placement of a storage-class specifier other than at the beginning of the declaration specifier in a declaration is an obsolescent feature.

Example:

```
int static i = 0;
```

Action:

Place the storage-class specifier first in the declaration.

Reference:

C99 6.7, 6.7.1

2102 forward declaration of enum type is nonstandard

Cause:

It is non standard to provide forward declarations for enumerations.

Example:

```
enum Consts;
```

Action:

Define the enumerations instead of forward declaration as: `enum Consts { MIN = 10, MAX = 100 };`

Reference:

ANSI/ISO C++ 7.2(1)

2108 signed bit field of length 1

Cause:

The compiler has detected the usage of a signed bit field of length 1. This can lead to unintuitive behavior as assigning 1 to such a bit-field value will make the value -1.

Example:

```
int field : 1;
```

Action:

Declare the 1-bit bit field as unsigned.

Reference:

2111 statement is unreachable

Cause:

Code at this location will never be executed. Often unreachable statements represent a real coding error such as a wrongly inserted return, goto, throw or call to non-returning functions etc.

Example:

```
return 0;
        i++; // warning 2111 here
```

Action:

Check for incorrectly placed unconditional control transfer statements.

Reference:

2117 non-void %n should return a value

Cause:

A function that returns a value does not end with a return statement. If function execution reaches the end of the function, the implied return statement that executes will return an undefined value.

Example:

```
int foo() {
        return;
}
```

Action:

End a non-void function with an appropriate return value.

Reference:

2120 return value type does not match the function type

Cause:

Value returned by the function does not match the return type of the function.

Example:

```
typedef char* caddr;
int foo() {
    caddr data = "0x1";
    return data;
}
```

Action:

Return a value that matches or is compatible with the return type of the function.

Reference:

2128 loop is not reachable from preceding code

Cause:

A loop has been detected by the compiler which can never be reached.

Example:

```
int main() {
    if (1)
        return 0;
    for(int a = 0; a < 3; a++) {
        if (a == 1)
            return 1;
    }
}
```

Action:

If the loop is necessary, change the code such that the loop can be reached. Otherwise, remove the loop.

Reference:

2144 a value of type %t1 cannot be used to initialize an entity of type %t2

Cause:

Initializing an entity of some type with incompatible types.

Example:

```
typedef char* addr;
int main() {
    addr data = "0x1";
    int bar = data;
    return 0;
}
```

Action:

Initialize an entity of some type with the same type or compatible types. Or use appropriate cast operator to convert one operand to the other.

Reference:

2147 declaration is incompatible with %nfd

Cause:

The declaration or definition provided is incompatible with a previous declaration.

Example:

```
class Init {
    static const float fmem;
};
float Init::fmem = 10.0f;
```

Action:

Check if given declaration/definition is consistent with the previous declaration. For example, the qualifiers might be missing, the type declared might be different etc. The given example can be corrected as follows: `class Init { static const float fmem; }; const float Init::fmem = 10.0f;`

Reference:

2152 conversion of nonzero integer to pointer

Cause:

The compiler has detected the conversion of a non-zero integer to pointer. Such conversions might lead to undesirable behavior.

Example:

```
#define DT 10
typedef char* caddr;
int main() {
    caddr data = "0x1";
    if (data == DT)
        return 1;
}
```

Action:

Avoid such incompatible type conversions.

Reference:

C99 6.3.2.3

2167 argument of type %t1 is incompatible with parameter of type %t2

Cause:

The parameter type passed is incompatible with the function argument type.

Example:

```
typedef char * addr;
int foo(int a) {
    return a;
}
int main() {
    addr data = "0x1";
    foo(data);
    return 0;
}
```

Action:

Pass the parameter type that is compatible with the function argument type.

Reference:

2170 pointer points outside of underlying object

Cause:

When a pointer to an object is not initialized, the offset might lie outside the base object. Hence, the behavior might be undefined.

Example:

Action:

Initialize the pointer.

Reference:

2172 external/internal linkage conflict with previous declaration %nod

Cause:

The linkage of a declaration conflicts with the linkage specified in an earlier declaration.

Example:

```
extern void bar();
static void bar() {}
```

Action:

Change one of the declarations so that the linkages match.

Reference: C99 6.2.2.7

2174 expression has no effect

Cause:

The compiler detected that the expression has no effect.

Example:

```
int main() {
    int index = 0;
```

```
        for(index; index < 10; index++){ }
    }
```

Action:

Modify the expression so that it has some effect or just remove it.

2175 subscript out of range

Cause:

The compiler has detected an array access that is outside the bounds of the array. The array access might cause unpredictable behavior.

Example:

```
int main() {
    int arr[10];
    arr[10] = 20;
}
```

Action:

Specify an array subscript that is within the bounds of the array type.

Reference:

2177 %n was declared but never referenced

Cause:

An identifier was defined, but never referenced in the translation unit.

Example:

```
int main() {
    int i;
    return 0;
}
```

Action:

Examine your code to determine if this definition is needed in this module.

Reference:

2180 argument is incompatible with formal parameter

Cause:

The type of the argument passed to the function in the function call differs from the declared type in the formal parameter.

Example:

```
int foo(int arg) {
    return arg;
}
int main() {
    foo("hello");
}
```

Action:

Ensure that the types of the argument passed matches with the types of formal parameters of the function.

Reference:

2181 argument is incompatible with corresponding format string conversion

Cause:

The compiler has detected a format specifier whose data type does not match its corresponding argument.

Example: `printf("%d", 2.0);`

Action:

Modify either the argument or the conversion specifier so that they match.

Reference: C99 6.2.2.7

2186 pointless comparison of unsigned integer with zero

Cause:

An unsigned expression is being tested to see if it is greater than zero. An ordered comparison between an unsigned value and a constant that is zero may indicate a programming error.

Example:

```
size_t i = 10;
    if(i >= 0)
        printf("true");
```

Action:

Cast (or otherwise rewrite) one of the operands of the compare to match the signedness of the other operand, or compare for equality with zero.

Reference:

2187 use of "=" where "==" may have been intended

Cause:

The assignment expression is used as the controlling expression of an if, while or for statement. A common user mistake is to accidentally use assignment operator "=" instead of the equality operator "==" in an expression that controls a transfer. For example saying if (a = b) instead of if (a == b). While using the assignment operator is valid, it is often not what is intended. When this message is enabled, the compiler will detect these cases at compile-time. This can often avoid long debugging sessions needed to find the bug in the user's program.

Example:

```
void check_ten(int arg) {
    if(arg = 10)
        printf("true");
}
```

Action:

Make sure that the assignment operator is what is expected.

Reference:

2188 enumerated type mixed with another type

Cause:

This warning is issued when mixing different enums or non-enums with enums.

Example:

```
enum Bool { FALSE = 0, TRUE = 1};
enum Bool b = FALSE + 1;
```

Action:

Make sure that you do not mix enums and non-enums.

Reference:

2191 type qualifier is meaningless on cast type

Cause:

Casting to a qualified type, though valid, is pointless. This is reported only when the cv-qualifiers are explicit in the cast, not, for example, when they are hidden in a typedef or template parameter type.

Example: `int i = (const volatile int) 10;`

Action:

Remove the unnecessary qualifiers in the cast.

Reference:

2192 unrecognized character escape sequence

Cause:

An escape sequence consists of a `'\'` (backslash) character followed by one or more characters. This diagnostic is issued if the character following the `'\'` is not known or not recognized by the compiler.

Example: `printf("%c", '\z');`

Action:

Check if the character provided after the `'\'` character is really a recognized escape sequence character.

Reference:

C99 5.2.1, 5.2.2

2193 zero used for undefined preprocessing identifier

Cause:

An identifier found in an `#if` or `#elif` is not defined. The compiler will replace the identifier with the constant zero.

Example:

```
int main() {
    #if MACRO
    return 0;
    #endif //MACRO
    return 1;
}
```

Action:

Verify the use of the identifier.

Reference:

2223 function %sq declared implicitly

Cause:

An expression contained a reference to a function that has not been declared. The C99 standard requires that all referenced functions must be declared before they are referenced.

Example:

```
int main() {
    foo(10);
}
```

```
}
```

Action:

Declare the function before it is referenced.

Reference:

C99 6.5.2.2

2224 the format string requires additional arguments

Cause:

The number of conversion specifiers is more than the number of values to be converted as specified in the parameter list.

Example: `printf("%d %d", 2);`

Action:

Make sure the number of conversion specifiers match the number of values to be converted.

Reference:

2225 the format string ends before this argument

Cause:

The number of conversion specifiers is less than the number of values to be converted as specified in the parameter list.

Example: `printf("%d ", 2, 2);`

Action:

Make sure the number of conversion specifiers match the values to be converted.

Reference:

2226 invalid format string conversion

Cause:

The compiler has detected an ill formed conversion specification (flags, width, precision, length modifier) or an unknown conversion specifier (not diouxefgcspn...) that will cause unpredictable behavior.

Example: `printf("%z ", 2);`

Action:

Review the documentation for this function and modify the conversion specification as appropriate.

Reference:

2228 trailing comma is nonstandard

Cause:

Accepting an enumerator list that contains a trailing comma is an extension of HP C provided for compatibility with other C compilers. An enumerator list with a trailing comma is not valid in C89, nor in C++. The C99 standard does permit this syntax.

Example:

```
typedef enum Switch {
    on,
    off,
} var;
```

Action:

Remove the trailing comma.

Reference:

C99 6.7.2.2 ANSI/ISO C++ 7.2

2231 declaration is not visible outside of function

Cause:

A type is declared within a function prototype. The type is local to the function prototype and will not be visible outside the prototype. This might cause unexpected errors later in the compilation.

Example: `void foo(struct X { int i; });`

Action:

Declare the type before the function prototype.

Reference:

2236 controlling expression is constant

Cause:

A boolean controlling expression tests for a constant pointer or pointer to member values. NOTE: This warning is not issued for cases like `if (1)`; since these often result from valid macro expansions.

Example:

```
void foo();
int main() {
    if (foo) { // warning 2236 here.
    }
}
```

Action:

A test of a constant address is unusual, and might be done by mistake - check if this is what you intended.

Reference:

2237 selector expression is constant

Cause:

The given expression in the controlling expression of a switch statement is a constant. Switch statements are usually used for variables to choose between various alternatives at runtime.

Example:

```
int main() {
    switch(10) {
        case 10: printf("ten\n"); break;
        case 20: printf("twenty\n"); break;
        default: printf("error\n");
    }
}
```

Action:

Check if it is a programming error.

Reference:

2245 a nonstatic member reference must be relative to a specific object

Cause:

Only static members can be accessed using the class name. For accessing non-static members, specific object must be used.

Example:

```
struct Dummy {
    int i;
};
Dummy::i = 10;
}
```

Action:

Access non-static members through objects only. For example: Dummy d; d.i = 10;

Reference:

2248 pointer to reference is not allowed

Cause:

Pointer to reference is incorrect and is not allowed.

Example:

```
int *i = 0;
int & * pri = i;
}
```

Action:

Check if you meant to declare a pointer to a reference. For example: int * & pri = i;

Reference:

ANSI/ISO C++ 8.3.2 (4)

2249 reference to reference is not allowed

Cause:

Reference to reference is incorrect and is not allowed.

Example:

```
int i = 0;
int & & ri = i;
}
```

Action:

Check if you typed extra & by mistake. For example: int & ri = i;

Reference:

ANSI/ISO C++ 8.3.2 (4)

2250 reference to void is not allowed

Cause:

Reference to void is incorrect and is not allowed.

Example: void & foo() { }

Action:

Remove the & and make it a plain void type. For example: void foo() {}

Reference:

ANSI/ISO C++ 8.3.2 (4)

2251 array of reference is not allowed

Cause:

Array of reference is incorrect and is not allowed.

```
Example: int & iarr[5];  
}
```

Action:

Remove the & and make it a plain array. For example: int iarr[5];

Reference:

ANSI/ISO C++ 8.3.2 (4)

2252 reference %n requires an initializer

Cause:

Reference variables should always be initialized, unless the reference is declared as extern , or it is declared as a class member within a class declaration, or it is declared as a parameter or function return type.

```
Example: int &ir;  
}
```

Action:

Provide an initializer for the reference. For example: int i; int &ir = i;

Reference:

ANSI/ISO C++ 8.3.2 (4)

2260 explicit type is missing ("int" assumed)

Cause:

The declaration has a storage-class specifier, but no type was specified. The compiler will assume the type of int. Omitting the type specifier is not valid in C++ or in C99, and is often considered poor programming practice.

```
Example: static i;
```

Action:

Add a type specifier to the declaration.

Reference:

C99 6.5.2.2, 6.7.2; C++ Section 7

2263 duplicate base class name

Cause:

The same base class name is specified more than once in the base class list. A base class name should appear only once in a base class list.

Example:

```
class Base {};  
class Derived: public Base, protected Base {};
```

Action:

Remove the redundant base class name, for example: class Derived: public Base {};

Reference:

2265 %nd is inaccessible

Cause:

An attempt is made to access a private or protected member in the context where the member is not accessible.

Example:

```
class A {
    private:
        int x;
    } c;
    c.x = 0;
```

Action:

Fix the access specifier of the member or the statement that is accessing the member.

Reference:

2267 old-style parameter list (anachronism)

Cause:

The given function is defined using the old style K&R syntax. The C standard has marked this syntax as obsolescent, and it is not supported in C++. Consider using the standard C prototype syntax.

Example:

```
int foo(arg)
    int arg;
{ return arg; }
```

Action:

Recode the function definition to use the recommended prototype-format definition.

Reference:

C99 6.9.1

2269 conversion to inaccessible base class %t is not allowed

Cause:

It is incorrect to assign a derived object to a protected or private base class pointer or reference without an explicit cast. NOTE: If an explicit access specifier is missing while inheriting from a base class then private inheritance is assumed. In case of inheriting from a struct public inheritance is assumed.

Example:

```
class Base {};
```

```
class Derived: protected Base{};
Base *b = new Derived;
```

Action:

Check if you actually want to convert from the derived to base type. If yes, then provide an explicit cast: `Base *b = (Base*) new Derived;`

Reference:

ANSI/ISO C++ 8.3.2 (4)

2276 name followed by :: must be a class or namespace name

Cause:

The qualifier type preceding `::` is not declared or defined as a class or a namespace.

Example: `T::type i;`

Action:

Make sure that the declaration or definition of the class or the namespace is defined before it is used. Check if the necessary header files containing the declaration/definition are included.

Reference:

2278 a constructor or destructor may not return a value

Cause:

Constructors and destructors do not have a return type and they do not return any value. It is incorrect to provide a return statement for constructors and destructors.

Example:

```
struct Base {
    Base() { return 0; }
};
```

Action:

Remove the return statement, for example: `Base() {}`

Reference:

2306 default argument not at end of parameter list

Cause:

Default arguments can only be provided at the end of the parameter list. All the parameters following a parameter with default argument must also have default arguments.

Example: `void print(int argc = 0, char **argv);`

Action:

Remove the given default arguments, or provide default arguments to all the parameters following that parameter or rearrange the parameter list. For example: `void print(int argc, char **argv); //` or `void print(int argc = 0, char **argv = 0); //` or `void print(char **argv, int argc = 0);`

Reference:

ANSI/ISO C++ 8.3.6 (4)

2314 only nonstatic member functions may be virtual

Cause:

Only non-static member functions can be declared as virtual, static members functions cannot be virtual.

Example:

```
class Base {
    virtual static void vfoo() {}
};
```

Action:

Make the member function either static or virtual, for example: `virtual void vfoo() {}`

Reference:

2315 the object has cv-qualifiers that are not compatible with the member function

Cause:

The cv qualifiers (const and volatile qualifiers) of the object should be compatible with the cv qualifiers of member functions that it calls. For example, any attempt to call a non-const member function on a const object will result in an error.

Example:

```
int main() {
    struct X {
```

```

        void foo();
    };
    const X s;
    s.foo();
}

```

Action:

Ensure that the cv qualifiers of the object are not stricter than cv qualifiers of the methods that called through it.

Reference:

ANSI/ISO C++ 9.3.2(4)

2319 pure specifier ("= 0") allowed only on virtual functions

Cause:

Only virtual functions can be declared as pure; other functions cannot have pure specifier as it is meaningless.

Example:

```

class base {
public:
    static int foo() = 0;
};

```

Action:

Provide the pure specifier only for virtual functions.

Reference:

ANSI/ISO C++ 9.3.1(4)

2320 badly-formed pure specifier (only "= 0" is allowed)

Cause:

Pure virtual functions are declared by a pure specifier indicated by = 0 ; = 0 does not indicate initialization or assignment, hence 0 cannot be replaced by other values.

Example:

```

class base {
public:
    virtual int foo() = 1;
};

```

Action:

Provide the pure specifier only as "= 0", for example: virtual int foo() = 0;

Reference:

ANSI/ISO C++ 9.2, 10.4(2)

2321 data member initializer is not allowed

Cause:

Only static const members of integral or enumeration type can be initialized inline in a class definition. For other types, provide the initialization outside the class definition.

Example:

```

int i = 100;
class Init {
    static const int *p_i =
};

```

Action:

Initialize the member outside the class: `int i = 100; class Init { static const int *p_i; }; const int* Init::p_i =`

Reference:

ANSI/ISO C++ 9.4.2 9.2(4)

2322 object of abstract class type %t is not allowed

Cause:

By definition, objects of abstract types cannot be created; any such attempt will be flagged as an error by the compiler.

Example:

```
class Base {
    virtual void foo() = 0;
};
Base b;
```

Action:

Either remove the statement that attempts to create an object of an abstract class type or make the class concrete/ non-abstract by removing the pure virtual functions.

Reference:

ANSI/ISO C++ 10.4(3)

2323 function returning abstract class %t is not allowed

Cause:

An abstract class cannot be used as a function return type, parameter type or as the type of an explicit conversion.

Example:

```
class Base {
    virtual void foo() = 0;
};

class Derived : public Base {
    Derived bar();
};
```

Action:

Do not declare a function to accept as parameter or return an abstract class type.

Reference:

ANSI/ISO C++ 10.4(3)

2325 inline specifier allowed on function declarations only

Cause:

The inline keyword is meaningful for function declarations only.

Example: `inline int i;`

Action:

Remove the inline keyword. For example: `int i;`

Reference:

ANSI/ISO C++ 10.4(3)

2329 local class member %n requires a definition

Cause:

Member functions of local classes must be defined within the class definition.

Example:

```
int main() {
    struct Struct {
        void foo();
    };
    Struct s;
    s.foo();
}
```

Action:

Define all required member functions of the local classes inside the class itself.

Reference:

ANSI/ISO C++ 9.8(2)

2336 unknown external linkage specification

Cause:

The linkage specification provided in the code is not known to this compiler implementation. The C and C++ linkage are two linkage specifications that a standard conformant compiler should support and linkage specifications for other languages are implementation defined.

Example: `extern "XYZ" int foo();`

Action:

Provide a valid linkage specification allowed by the compiler, like C or C++ , for example:

`extern "C" int foo();`

Reference:

ANSI/ISO C++ 7.5 (3)

2340 value copied to temporary, reference to temporary used

Cause:

References should be initialized to refer to a valid object or a function. In certain contexts, when rvalues are used for initializing references, the compiler might create a temporary and bind that to the reference. In such scenarios compiler will issue this diagnostic.

Example:

```
struct Init {
    const int & mem;
    Init() : mem(10) {}
};
```

Action:

Provide a proper object for initializing the reference.

Reference:

ANSI/ISO C++ 8.5.3 12.2

2363 invalid anonymous union -- nonpublic member is not allowed

Cause:

Anonymous unions can have only public non-static data members, private or protected members are not allowed.

Example:

```
static union {
    private:
        int i;
    protected:
        float f;
};
```

Action:

Provide public access to all the anonymous union members. For example:

```
static union {
    public:
        int i;
        float f;
};
```

Reference:

ANSI/ISO C++ 9.5(3)

2364 invalid anonymous union -- member function is not allowed

Cause:

Anonymous unions cannot have member functions, they can only contain non-static data members.

Example:

```
static union {
    int foo() { return 0; }
};
```

Action:

Remove the member function from anonymous union.

Reference:

ANSI/ISO C++ 9.5(3)

2365 anonymous union at global or namespace scope must be declared static

Cause:

Anonymous unions at global or namespace scope should be qualified with static qualifier to limit them to file scope.

Example:

```
union {
    int i;
};
```

Action:

Add static storage class to the anonymous union declared at global or namespace scope. For example:

```
static union {
    int i;
};
```

Reference:

ANSI/ISO C++ 9.5(3)

2375 declaration requires a typedef name

Cause:

In a typedef declaration, the typedef name is missing. This case can occur when a declaration tries to declare both a tag and a typedef, but the name of the typedef is not included.

Example: `typedef int;`

Action:

Either remove the `typedef` keyword or add a typedef name.

Reference:

2381 extra ";" ignored

Cause:

An extra semicolon was found at the end of some code construct, which was perhaps added by mistake. It will be ignored.

Example: `int main() {};`

Action:

Remove the extra semicolon.

Reference:

2487 inline %n cannot be explicitly instantiated

Cause:

When explicit instantiation of a class template is done, inline functions will not be instantiated. This diagnostic is just to mention this fact.

Example:

```
template <typename T>
class X {
    void foo() {}
};

template class X<int>;
```

Action:

Providing explicit instantiations for templates is a old programming style and it is not recommended. Check if you really need to do explicit instantiations in your code.

Reference:

ANSI/ISO C++ 14.7.2

2513 a value of type %t1 cannot be assigned to an entity of type %t2

Cause:

The types of variables used in the assignment are incompatible.

Example:

```
long double d = 0.0;
char *c = 0;
c =
```

Action:

Check if the assignment is required and if it is valid. If valid, use an explicit cast to force the assignment.

Reference:

2546 transfer of control bypasses initialization of: "variable"

Cause:

Compiler has detected transfer of control was attempted that bypasses initialization of a variable.

Example:

```
int main() {
    goto LABEL;
    int i = 20;
LABEL:
    printf("jumped to LABEL\n");
}
```

Action:

Rewrite the code such that declarations/initializations are not provided within the code segment which can be bypassed because of explicit transfer of control.

Reference:

2549 "variable" is used before its value is set

Cause:

A variable's value has been used without being set. The algorithms that detect this situation only report it once for a given variable, and not necessarily at the first use of the uninitialized value.

Example:

```
int i;
    printf("%d", i);
```

Action:

Provide the variable with a value before the variable is used. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected. It is best to initialize variables as close as possible to the point of declaration.

Reference:

2550 %n was set but never used

Cause:

This identifier is set but never referenced in the current translation unit.

Example:

```
int main() {
    int var;
    var = 20;
}
```

Action:

Examine your code to determine if this definition is needed in this module.

Reference:

2656 transfer of control into a try block

Cause:

Goto statements cannot be used to transfer control inside the try blocks.

Example:

```
goto LABEL;
    try {
        LABEL:
            throw 0;
    } catch(...) {
    }
```

Action:

Remove the goto statement that transfers the control inside the try block.

Reference:

ANSI/ISO C++ 15(2)

2767 conversion from pointer to smaller integer

Cause:

The 64-bit pointer is being cast to an integer type that is smaller in size. Casting a 64-bit pointer to a smaller integer type is undefined behavior. This also could indicate code that relies on pointers and integers being the same size. The code will cause an unexpected loss of data on 64-bit platforms.

Example:

```
int *p = 0;
int i = (int) p;
```

Action:

If this is the intended behavior, first cast the pointer to a 64-bit integer, then cast the result to the desired integer type.

Reference:

2815 type qualifier on return type is meaningless

Cause:

A type qualifier has been used as part of a function's return type. The type qualifiers have no meaning for function return values.

Example: `const int foo() { return 10; }`

Action:

Remove the type qualifier in the return type of the function.

Reference:

2826 %n was never referenced

Cause:

This function definition contains a parameter `n` that was never referenced.

Example: `int foo(int i) {
 return 0;
}`

Action:

Examine your code to determine if this function parameter is needed in this function.

Reference:

2830 %n has no corresponding operator delete%s (to be called if an exception is thrown during initialization of an allocated object)

Cause:

For operator new, a corresponding operator delete needs to be provided so that it can be called for proper destruction in case an exception is thrown during the initialization of the allocated object.

Example:

```
void* operator new(unsigned long, void*) { return 0; }
struct Pool {
    Pool() { }
} object;

Pool* allocate() { return new (&object) Pool(); }
```

Action:

Provide a corresponding operator delete to be called if an exception is thrown during the initialization of the allocated object.

Reference:

2836 returning reference to local variable

Cause:

It is incorrect to return a reference to a local variable as the return value from a function. This is because the lifetime of a local variable ends once the function returns.

Example:

```
int& foo() {
    int i = 10;
    return i;
}
```

Action:

Do not return reference to a local variable; if you really want to return a reference, make the local variable static. This is because the lifetime of a static variable is same as lifetime of the program.

Reference:

2837 omission of explicit type is nonstandard ("int" assumed)

Cause:

A declaration or a definition does not have an explicit type, int will be assumed as the type but this behavior is non-standard.

Example:

```
extern i;
a();
b(){}
extern c();
extern d(){}
```

Action:

Add an explicit type for the declaration or definition.

Reference:

C99 6.7.2(2), ANSI/ISO C++ 7.1.5(2)

2940 missing return statement at end of non-void function "function"

Cause:

A function with an explicit return type does not contain a return statement.

Example: `int foo() { }`

Action:

If the function is supposed to return a value, add a return statement with appropriate value, otherwise declare the return type as void.

Reference:

2951 return type of function "main" must be int

Cause:

Standard C requires that the return type of "main" function must int. In case of non int return type the status value returned to the environment may not be what you expect. Also, this code is not portable.

Example: `long main() { }`

Action:

Define the "main" function with a return type of int for maximal portability.

Reference:

C99 5.1.2.2.1

2991 extra braces are nonstandard

Cause:

An initializer contains extra open or close braces for the object being initialized.

Example: `int arr [4] = { { 0 } };`

Action:

Remove extra braces.

Reference:

3000 a storage class may not be specified here

Cause:

A storage class can only be specified for an object or a function.

Example: `static enum nothing { nil = 0 };`

Action:

Remove the unnecessary storage class specifier.

Reference:

ARM 7.1.1.

3051 standard requires that %n be given a type by a subsequent declaration (\ "int\" assumed)

Cause:

The parameter of an old-style function definition was not declared. It will default to int type. Omitting the type specifier is not valid in C99.

```
Example: int foo(arg) { return arg; }
```

Action:

Declare the parameter. Preferable old-style function definitions should be replaced by prototype-format definitions.

Reference:

C99 6.9.1

3055 types cannot be declared in anonymous unions

Cause:

Anonymous unions can only define non-static data members. You cannot declare types or functions within anonymous unions.

Example:

```
static union {
    typedef int T;
    T i;
};
```

Action:

Remove the typedef from the anonymous union. Either declare the members without the typedef type or provide the typedef outside the anonymous union. For example: `static union { int i; };`

Reference:

ANSI/ISO C++ 9.5(2)

3056 returning pointer to local variable

Cause:

The return value of the function is the address of a local variable. Unless declared as static, a local variable has automatic storage duration i.e. the storage for it lasts until the block that defines it exists. Dereferencing the pointer to a local variable after the function that defines it returns will lead to undefined runtime behavior.

Example:

```
int* foo() {
    int i = 10;
    return
}
```

Action:

If you need to return the pointer to this variable then make it a static variable.

Reference:

ANSI/ISO C++ 3.7.2

3105 #warning directive: %s

Cause:

The preprocessor has encountered a #warning preprocessor directive.

Example:

```
#if VERSION_10
#define VERSION 10
#else
#warning "undefined version"
#endif
```

Action:

#warning directives are usually added to warn against fallthrus in conditionally compiled code. check if you need to handle a new condition.

Reference:

3138 format argument does not have string type

Cause:

The format argument of a function having same attributes as the printf, scanf family functions must be of string type.

Example:

```
int print(char *str, void *fmt, ...)
    __attribute__((format(printf, 2, 3))); // argument 2 must be of
                                         // string type
```

Action:

Change the type of format argument to string type

Reference:

3145 %t1 would have been promoted to %t2 when passed through the ellipsis parameter; use the latter type instead

Cause:

Arguments that are passed as variable arguments are subjected to default argument promotion. The programmer should be aware of such promotions and should ensure that they do not effect the correctness of the program.

Example:

```
int print(char *str, void *fmt, ...)
    __attribute__((format(printf, 2, 3))); // argument 2 must be of
                                         // string type
```

Action:

Where possible use the promoted type instead

Reference:

ANSI/ISO C++ 5.2.2(7)

3197 the prototype declaration of %nfd is ignored after this unprototyped redeclaration

Cause:

The unprototyped declaration of a function in an inner scope hides an prototyped declaration in the outer scope. NOTE: In same scope the function prototypes do not hide each other but in nested scope they do.

Example:

```
int foo(int x);
int main() {
    int foo(); // warning 3197 here
}
```

Action:

Remove the unprototyped declaration.

Reference:

3290 Passing a non-POD object to a function with variable arguments has undefined behavior. Object will be copied onto the stack instead of using a constructor.

Cause:

A non-POD (POD stands for "plain old data") object is being passed to a function with variable arguments. Object will be copied onto the stack instead of using a constructor. Varargs do not know how to deal with non-POD types, this can lead to undefined behavior.

Example:

```
class A {
    int i;
};

int foo(char*, ...);
int bar() {
    A a;
    foo("hello", a);
    return 0;
}
```

Action:

Do not pass non-POD object to a function with variable arguments. If this is essential write a conversion operator to convert from non-POD to POD type and call that operator before passing the object to the variable arguments function.

Reference:

3348 declaration hides %nd

Cause:

A declaration hides another variable with the same name visible in this scope.

Example:

```
struct X {
    X(int i) {
        i = i;
    }
    int i;
};
```

Action:

Examine your code to see if any of the two declarations should be removed/modified.

Reference:

3353 %n has no corresponding member operator delete%s (to be called if an exception is thrown during initialization of an allocated object)

Cause:

For member operator new, a corresponding member operator delete needs to be provided so that it can be called for proper destruction in case an exception is thrown during the initialization of the allocated object.

Example:

```
struct Pool {
    Pool() { }
```

```
void* operator new(unsigned long, void*) { return 0; }  
} object;
```

Action:

Provide a corresponding member operator delete to be called if an exception is thrown during the initialization of the allocated object.

Reference:

3750 "\" followed by white space is not a line splice

Cause:

This warning is issued when a space is found after \ (backslash) at the end of line.

Example:

```
int main()  
  
    {  
    int something; \  
    int anotherthing;  
    }
```

Action:

A space after \ (backslash) at the end of line is unusual, and might be done by mistake - check if this is what you intended.

Reference:

4212 mismatch between character pointer types %t1 and %t2

Cause:

C allows conversion between pointers to interchangeable types, but this can be unsafe. Mismatch between character pointer types is diagnosed with a unique message so it can be suppressed if desired.

Example:

```
int foo(char *x, unsigned char *y) {  
    x = y; // warning 4212 here  
}
```

Action:

If this conversion is safe for your code, then suppress this warning by providing an explicit cast.

Reference:

C99 6.3.2.3

4225 suggest parentheses around comparison in operand of %sq

Cause:

The bitwise operator | has lower precedence than the comparison operator, possibly changing the semantics of the expression. This can be caused by other combinations of comparison and bitwise operators as well.

Example:

```
bool check(unsigned p1, unsigned p2, unsigned mask) {  
    return ( p1 == p2 | mask );  
}
```

Action:

Disambiguate the expression by using explicit parenthesis around the bitwise operator, thus return (p1 == (p2 | mask));

Reference:

4227 padding struct with %s1 bytes to align member %sq2

Cause:

The compiler has added s1 bytes before a member so that it will be aligned and hence accessed efficiently.

Example:

```
typedef struct {
    int field1;
    double field2;
} str;
```

Action:

Insertion of padding bytes themselves is not a problem but you need to ensure that you do not use hard coded offsets for accessing fields of the struct through pointers, use offset of macro instead. In some cases the number of padding bytes being inserted can be reduced by reordering the fields.

Reference:

4228 64 bit migration: conversion from %t1 to a more strictly aligned type %t2 may cause misaligned access

Cause:

The compiler has detected conversion of pointers from a lesser aligned type to a more strictly aligned type. This usually happens when assigning an int pointer to a long pointer. This is not a problem in 32 bit mode since int and long are both 4 bytes aligned but in 64 bit mode int is 4 byte aligned whereas long is 8 bytes aligned. Because of the difference in alignment in 64 bit mode this conversion might cause unaligned access in 64 bit mode.

Example:

```
int a = 10;
long *lptr = (long *)&a;
```

Action:

For 64 bit portability ensure that your code does not assign a pointer to int to a pointer to long.

Reference:

4229 64 bit migration: conversion from %t1 to %t2 may truncate value

Cause:

The compiler has detected conversion between pointers to data types having different sizes. This usually happens when assigning an int pointer to a long pointer or vice versa. This is not a problem in 32 bit mode since int, long and pointer all are of same size (32 bit) but in 64 bit mode int is 32 bit whereas long and pointer are 64 bit. Because of the difference in size in 64 bit mode this conversion might lead to incorrect behavior in 64 bit mode.

Example:

```
void foo(long l) {
    int i = l;
}
```

Action:

For 64 bit portability ensure that your code does not convert from pointer and long values to int.

Reference:

4230 64 bit migration: conversion from %t1 to %t2 may cause target of pointers to have a different size.

Cause:

The compiler has detected conversion between pointers to data types having different sizes. This usually happens when assigning an int pointer to a long pointer or vice versa. This is not a problem in 32 bit mode since int, long and pointer all are of same size (32 bit) but in 64 bit mode int is 32 bit whereas long and pointer are 64 bit. Because of the difference in size in 64 bit mode this conversion might lead to incorrect behavior in 64 bit mode.

Example:

```
long l = 1;
long *lptr = &l;
int *iptr = (int *)lptr;
```

Action:

For 64 bit portability ensure that your code does not convert from pointer to int to pointer to long and vice versa.

Reference:

4231 64 bit migration: conversion between types of different sizes has occurred (from %t1 to %t2)

Cause:

The compiler has detected conversion between data types having different sizes. This usually happens when converting from int data type to pointer or long data type and vice versa. This is not a problem in 32 bit mode since int, long and pointer all are of same size (32 bit) but in 64 bit mode int is 32 bit whereas long and pointer are 64 bit. Because of the difference in size in 64 bit mode this conversion might lead to incorrect behavior in 64 bit mode.

Example:

```
void foo(int i) {
    foo((void*)i);
    return i;
}
```

Action:

For 64 bit portability ensure that your code does not convert from int to pointer or long and vice versa.

Reference:

4232 conversion from %t1 to a more strictly aligned type %t2 may cause misaligned access

Cause:

A pointer is being cast from a lesser aligned type to a more strictly aligned type. Accesses using the new type may cause misaligned access.

Example:

```
int foo(int* p, long long* l) {
```

```
        l = (long long*) p; // warning 4232 here
    }
```

Action:

Correct the code that uses such conversions.

Reference:

ANSI/ISO C++ 3.9, 3.9.1, 3.9.2

4235 conversion from %t1 to %t2 may lose significant bits

Cause:

The destination of an assignment has less range/precision than the source operand, this might cause a loss of value/precision.

Example:

```
int i = 10;
double l = 10.345;
int j = l/i;
```

Action:

Change the type of the destination or if the loss in significant bits is not a concern add a cast.

Reference:

4237 type cast from %t1 to %t2 may cause sign extension to a larger size integer.

Cause:

A signed integer constant is being converted to a larger unsigned integral type. The conversion might cause sign # extension leading to a larger value than expected.

Example:

```
typedef unsigned long long uint64_t;
int i = -1;
uint64_t a = (uint64_t) i;
```

Action:

Verify if the type cast behavior is intended.

Reference:

4239 case type mismatch with switch expression type

Cause:

The switch expression type does not match with the type of case labels.

Example:

```
enum BOOLEAN { TRUE = 0, FALSE = 1 };
BOOLEAN a_bool = TRUE;
switch(a_bool) {
    case 0: printf("true\n");
    case 1: printf("false\n");
    default: printf("no match found \n");
}
```

Action:

Fix your source code to ensure the type of the switch expression and case labels is same.

Reference:

4241 redeclaration, function %nod was previously called without a prototype

Cause:

The switch expression type does not match with the type of case labels.

Example:

```
enum BOOLEAN { TRUE = 0, FALSE = 1 };
BOOLEAN a_bool = TRUE;
switch(a_bool) {
    case 0: printf("true\n");
    case 1: printf("false\n");
    default: printf("no match found \n");
}
```

Action:

Fix your source code to ensure the type of the switch expression and case labels is same.

Reference:

4242 No prototype or definition in scope for call to %sq

Cause:

Prototypes are optional in ANSI C, however their use can prevent a wide range of common programming errors which are otherwise difficult to detect. This call site provides no prior prototype or definition so the compiler cannot verify the correctness of the argument list.

Example:

```
int main() {
    foo(10.0f);
}
```

Action:

Provide a prototype declaration for the function before calling that function.

Reference:

4243 function declared with empty parentheses, consider replacing with a prototype

Cause:

The declaration of function has an empty parameter list. If the function has parameters, they should be declared here; if it has no parameters, "void" should be specified in the parameter list.

Example: `int foo();`

Action:

The recommended way to declare a function that takes no parameters is to use "void" in the parameter list.

Reference:

4244 extern storage class used with a function definition

Cause:

The compiler has detected the usage of an extern storage class specifier with a function definition.

Example:

```
extern int foo() {
```

```
        return 0;
    }
```

Action:

Remove the extern storage class specifier in function definition.

Reference:

4245 storage class used with a data definition

Cause:

The compiler has detected the usage of an extern storage class specifier with a data definition.

Example: `extern int i = 1;`

Action:

Remove the extern storage class specifier in the data definition.

Reference:

4247 function called with different argument counts (%s vs. %s2)

Cause:

Two calls to an unprototyped function differ in the number of arguments being passed, at least one of these calls is passing wrong arguments.

Example:

```
int main() {
    foo(10);
    foo(10, 100);
}
```

Action:

Correct the function call which is passing wrong number of arguments. Preferably declare the function before calling it.

Reference:

4248 comparison of unsigned integer with a signed integer

Cause:

This occurs due to mixing signed and unsigned operands with relational operators $<$, \leq , $>$, \geq . One of the operands is signed and the other unsigned. The signed quantity may be negative as well.

Example:

```
int a = -1;
unsigned int b = 1;
if (a < b)
    return a;
```

Action:

To resolve this problem, either cast the integer to unsigned if you know it can never be less than zero or cast the unsigned to a signed int if you know it can never exceed the maximum value of int.

Reference:

4249 64 bit migration: value could be truncated before cast to bigger sized type.

Cause:

The size of the result of an operation is determined based on the size of the operands and not based on the type into which the resultant value is being casted. If the result exceeds the size of the operands, in this case 32 bits, it will be truncated even before the cast takes place.

Example:

```
void foo(int data) {
    long data1 = (long) (data << 16); // warning 4249 here
    long data2 = ((long)data) << 16; // no warning
}
```

Action:

If you expect the value of the result to exceed the size of the type of operands then cast the operands to the bigger type before the operation.

Reference:

4251 the assignment has an excessive size for a bit field

Cause:

The constant has a larger value than that can be stored in the bit-field.

Example:

```
struct A {
    unsigned int f1 : 1;
};
int main() {
    A obj;
    obj.f1 = 30;
}
```

Action:

Ensure that the assigned value is within the range of the values that the bit-field can store.

Reference:

4253 unsigned value cannot be less than zero

Cause:

An ordered comparison between an unsigned value and a constant that is less than or equal to zero often indicates a programming error. A negative value is converted to an unsigned value before the comparison, so any negative value compares larger than most unsigned values. If the code is correct, the comparison could be more clearly coded by testing for equality with zero.

Example:

```
int main() {
    unsigned data = 30;
    if(data <= 0)
        return 1;
}
```

Action:

Cast (or otherwise rewrite) one of the operands of the compare to match the signedness of the other operand, or compare for equality with zero.

Reference:

4255 padding size of struct %sq1 with %s2 bytes to alignment boundary

Cause:

Padding bytes have been inserted for proper alignment of the struct.

Example:

```
typedef unsigned char uint8_t;
typedef unsigned int uint32_t;
int main() {
    struct X {
        uint32_t data_size;
        uint8_t status;
    };
}
```

Action:

Insertion of padding bytes themselves is not a problem but you need to ensure that you do not use hard coded offsets for accessing fields of the struct through pointers, use offset of macro instead. In some cases the number of padding bytes being inserted can be reduced by reordering the fields.

Reference:

ANSI/ISO C++ 18.1(5)

4259 suggest parentheses around the operands of %sq

Cause:

A possibly incorrect combination of [in]equality and bitwise operations. Passing a boolean argument to a bit operation is quite unusual and usually happens because of programming errors like missing parentheses around operands of "&" and "|"

Example:

```
return (i == j | k); // warning 4259 here
== has higher precedence than | and hence will be evaluated first, this is probably not what the programmer
intended. For j|k to be evaluated first put parentheses around j|k.
```

Action:

When using bit operations in combination with [in]equality, put parentheses around operands of bitwise operators.

Reference:

4264 padding size of struct anonymous with %s bytes to alignment boundary

Cause:

Padding bytes have been inserted for proper alignment of the anonymous struct.

Example:

```
typedef unsigned char uint8_t;
typedef unsigned int uint32_t;
int main() {
    typedef struct {
        uint32_t data_size;
        uint8_t status;
    } str;
}
```

Action:

Insertion of padding bytes themselves is not a problem but you need to ensure that you do not use hard coded offsets for accessing fields of the struct through pointers, use offset of macro instead. In some cases the number of padding bytes being inserted can be reduced by reordering the fields.

Reference:

4272 conversion from %t1 to %t2 may lose sign

Cause:

The Compiler has detected conversion from a signed data type to an unsigned data type. This may lead to loss of sign.

Example:

```

typedef unsigned int uint32_t;
void foo(int j) {
    uint32_t k;
    k = j;
}

```

Action:

If this is intended then add a cast.

Reference:

4273 floating-point equality and inequality comparisons may be inappropriate due to round off common in floating-point computation

Cause:

Since floating point numbers are usually subject to rounding-off errors, comparison between non-constant floating point values may not always be accurate.

Example:

```

bool check(float v1, float v2) {
    return ( v1 == v2 );
}

```

Action:

Change your program logic to not depend on such comparisons.

Reference:

4274 comparison of pointer with integer zero

Cause:

A pointer is being compared with an integer zero. Equality comparison with integer zero is allowed for checking if the pointer is null but other comparisons like $>$, $<$, \geq and \leq are not valid.

Example:

```

int main() {
    char* ptr = 0;
    if(ptr >= 0)
        return 1;
}

```

Action:

Rewrite the expression to use $==$ or $!=$.

Reference:

4275 constant out of range (%s) for the operator

Cause:

For a given relational operation the constant is out of range specified by the other non-constant operand.

Example:

```
void foo(bool b, char c) {
    if (b == 3) { } // warning 4275 here
    if (c < 200) { } // warning 4275 here
    if ((unsigned char)c < 200) { } // no warning here
}
```

Action:

Check the value of the constant being used in the operation. In the scenarios where the comparison is between signed and unsigned types the problem can be fixed by using a cast.

Reference:

4276 relational operator %sq always evaluates to 'false'

Cause:

The greater than or less than relational operation in the expression always evaluates to false. This happens when comparing against the largest and smallest values in the range of given type.

Example:

```
void foo(unsigned char c) {
    if (c > 255) { } // warning 4276 here
}
NOTE: 255 is the largest value for an unsigned char so a >
comparison will always be false.
```

Action:

Check the value of the constant being used in the operation.

Reference:

4277 logical AND with a constant, do you mean to use '&'?

Cause:

Using a constant in a logical AND (&&) operation is rather unusual. This usually indicates a typo where the programmer actually meant to say bitwise AND (&). This warning is not generated for the constant 0 since that often results from macro expansions.

Example: `result = value && 0x1; // warning 4277 here`

Action:

Check if you meant to use '&' instead of '&&'. For certain macro expansions this can be valid code, suppress this warning for those macros using the +Wmacro option.

Reference:

4278 the sub expression in logical expression is a constant

Cause:

Logical operators follow short-circuit evaluation - if the first operand of a logical || or && operator determines the result of the expression, the second operand will not be evaluated. In the given expression the first sub expression is a constant that determines the result of the expression and hence the rest of the expression will never be evaluated.

Example:

```
int main() {
    int a = 10;
    if( 10 || (a == 10))
        return 0;
}
```

Action:

Check if the constant is expected. In some cases the constant value might be the result of a valid macro expansion, in such scenarios you can suppress this warning for the particular macro using the `+Wmacro` option.

Reference:

4279 the expression depends on order of evaluation

Cause:

The expression modifies a variable more than once without an intervening sequence point OR the expression modifies a variable and fetches its value in a computation that is not used to produce the modified value without an intervening sequence point. The final value of such expressions is undefined.

Example:

```
i = i++; // warning 4279 here
x = i + ++i; // warning 4279 here
i = i + 1; // no warning here
```

Action:

Rewrite the expression so that each variable is modified only once before a sequence point OR if a variable is modified, it is fetched only to compute the value to be stored in the variable (ex `i = i+1`).

Reference:

C99 6.5, ANSI/ISO C++ 5.4

4281 assignment in control condition

Cause:

An assignment operator `=` was found where an equality operator `==` might have been intended, ex in an `if` condition expression. While using the assignment operator is valid such expressions often turn out to be bugs in the user's program.

Example: `if (a = b) ...`

Action:

Check if the assignment operator is what was intended.

Reference:

4286 return non-const handle to non-public data member may undermine encapsulation

Cause:

A less-restrictive member function is returning a non-const reference or pointer to a more-restrictive data member. Since the handle thus returned is not a `const`, the caller will be able to modify the restrictive data member, thus violating the norms of encapsulation.

Example:

```
class Capsule {
public:
    int& getvalue_ref() { return private_data; }
    int* getvalue_ptr() { return }
private:
    int private_data;
```

```
} object;
```

Action:

Declare the member functions to return a const handle instead.

Reference:

4289 endian porting: the definition of the union may be endian dependent

Cause:

The values accessed using the members of a union may differ based on endianness of a machine in which code is executed. This occurs when the union has members of different sizes, and when the value stored using one member is accessed using another of a different size.

Example:

```
union Endian { char c[4]; int v; }; // warning 4289 here
int foo() {
    union Endian u = { 0x11, 0x22, 0x33, 0x44 };
    int i = u.v;
    return i;
}
// On a big endian machine value of i will be 0x11223344 whereas
// on a little endian machine it will be 0x44332211
```

Action:

For portable code change the way this union is used. Do not use unions for converting values from one type to another.

Reference:

4290 endian porting: the initialization of char array may be endian dependent

Cause:

A char array is being initialized using hexadecimal values. It is likely that this array will later be accessed as an integer. Based on endianness of a machine on which code is executed the value of this integer will differ.

Example: `char a[4] = { 0x11, 0x22, 0x33, 0x44 }; // warning 4290 here`

Action:

For portable code check how this array is used.

Reference:

4292 endian porting: the dereference of cast pointer may be endian dependent

Cause:

A pointer of integral type is being cast to a pointer of char or smaller integer type. Based on endianness of a machine on which code is executed the value accessed using will differ.

Example:

```
unsigned int value = 0x03020100;
unsigned int *ptr =
unsigned char charVal;
void foo() {
    charVal = *(unsigned char *)ptr; // warning 4292 here
    charVal = *(unsigned char *) (void*)ptr; // no warning
}
```

Action:

For portable code change the way these values are accessed. If the order in which the values are accessed is not important then first cast the pointer to `void*` type and then cast to the lesser integer type.

Reference:

4295 abstract function type declared with empty parentheses, consider replacing with parameter list or void.

Cause:

The declaration of function has an empty parameter list. If the function has parameters, they should be declared here; if it has no parameters, `void` should be specified in the parameter list.

Example: `void foo(long (*compar) ()) ;`

Action:

The recommended way to declare a function that takes no parameters is to use `void` in the parameter list.

Reference:

4298 64 bit migration: addition result could be truncated before cast to bigger sized type

Cause:

The size of the result of an operation is determined based on the size of the operands and not based on size of variable in which the resultant value is stored. If the result exceeds the size of the operands, in this case 32 bits, it will be truncated even before the assignment takes place.

Example:

```
void foo(int data) {
    long data1 = data + 16; // warning 4298 here
    long data2 = ((long)data) + 16; // no warning
}
```

Action:

If you expect the value of the result to exceed the size of the type of operands then cast the operands to the bigger type before addition else use same type for operands and result.

Reference:

4299 64 bit migration: multiply result could be truncated before cast to bigger sized type

Cause:

The size of the result of an operation is determined based on the size of the operands and not based on size of variable in which the resultant value is stored. If the result exceeds the size of the operands, in this case 32 bits, it will be truncated even before the assignment takes place.

Example:

```
void foo(int data) {
    long data1 = data * 16; // warning 4299 here
    long data2 = ((long)data) * 16; // no warning
}
```

Action:

If you expect the value of the result to exceed the size of the type of operands then cast the operands to the bigger type before multiplication else use same type for operands and result.

Reference:

4300 Overflow while computing constant in left shift operation

Cause:

An overflow occurred while evaluating a constant in a left shift operation.

Example:

```
#define COUNT 24
#define ALPHA 0x0000EF
int main() {
    int a = ALPHA << COUNT;
}
```

Action:

Correct the constant expression so that it does not overflow.

Reference:

4301 expression has no effect

Cause:

The expression has no effect or side-effect.

Example:

```
#define foo(A) 0
int main() {
    int A = 0;
    foo(A);
}
```

Action:

Remove or correct the expression. In some cases the expression might be the result of a valid macro expansion, in such scenarios you can suppress this warning for the particular macro using the `+Wmacro` option.

Reference:

4314 if statement without body, did you insert an extra ';'?

Cause:

The if statement does not have a body. No action is performed when the if condition is met.

Example:

```
int main() {
    if (1) ;
}
```

Action:

Check if you inadvertently inserted a ';'.

Reference:

4315 %s loop without body, did you insert an extra ';'?

Cause:

The loop statement does not have a body.

Example: `for (i=0; i<10; i++) ; // warning 4315 here`

Action:

Check if you inadvertently inserted a ';'.

Reference:

4354 One of the operands of the %sq operation is a string literal, strcmp() is recommended for such comparison

Cause:

If the character pointer is compared with a string literal, then the result is unspecified.

```
char *ch_ptr = "hello";
if(ch_ptr == "world") { }
```

Action:

Use of '==' to compare 2 strings is unusual and might be done by mistake - check if you should be using strcmp() instead.

Reference:

4355 the initializer for %n is greater than %s

Cause:

This warning is issued when a large sized array (greater than 32 MB) is getting initialized in the program.

```
enum try_stuff {try_array= 0x08000000};
char try[try_array*4] = {1,2,[try_array*4-1]=10 };
```

Action:

It is safe to create an array of size less than 32 MB.

Reference:

4356 operand of sizeof is a constant rvalue, this might not be what you intended

Cause:

This warning is issued for sizeof(x) where x is a constant rvalue.

Example:

```
#include <stdio.h>
#define CONST 10
int foo()
{
    printf("%u\n", (unsigned) sizeof(10)); // Issue warning
    printf("%u\n", (unsigned) sizeof(CONST)); // Issue warning
    return 0;
}
```

Action:

Using sizeof() on a constant is unusual, and might be done by mistake - check if this is what you intended.

Reference:

4357 octal escape sequence "%s" is followed by decimal character '%s2'

Cause:

When a non-octal decimal character follows an octal sequence, you may assume that its part of the escape sequence.

Example:

```
int main() {
```

```

    const char *ch0 = "\028\4471";
    return 0;
}

```

Action:

For example: "\028" - Here '8' follows escape sequence '\02'. You can change the string to "\02" "8".

Reference:

4360 size of types of consecutive bitfields are different, bitfield packing behavior may be different across compiler

Cause:

This warning is issued during bitfield packing when the types in the structure are different.

Example:

```

#include<stdio.h>
struct node1
{
    bool b1:1;
    short s:1;
    bool b:1;
}head1;

int main()
{
    printf ("sizeof struct node1 is : %lu\n", sizeof(head1));
    return 0;
}

```

Action:

It is good to use similar types for bitfield packing. This is because, bitfield packing behavior might be different across compilers.

Reference:

4361 64 bit migration: size of types of consecutive bitfields are different, bitfield packing behavior may be different across compilers

Cause:

In a 32-bit to 64-bit migration scenario, this warning is issued during bitfield packing when the types in the structure are different.

Example:

```

#include<stdio.h>
enum name {first, middle, last};

struct node2
{
    long l:1;
    name n:1;
}head2;

int main()
{
    printf ("sizeof struct node2 is : %lu\n", sizeof(head2));
    return 0;
}

```

Action:

It is good to use similar types for bitfield packing. This is because, bitfield packing behavior might be different across compilers.

Reference:

4363 possible mismatch in parameters passed to `%n`, previously seen `%p`

Cause:

During compilation of C programs, when a call is made to a function without prototype, the compiler does not know the exact parameter types. It tries to get this information from the actual parameters being passed in the call. This warning is issued when there is a mismatch in the parameter types between different call.

Example:

```
File 1.c
int main()
{
    int i, j;

    I = ggg(10);
    j = ggg(10.3);
}
```

```
File 2.c
int ggg(long a)
{
    return 0;
}
```

Action:

Add prototypes for all external functions that is called in the file.

Reference:

4364 endian porting: type cast is endian dependent

Cause:

This warning is issued for pointer casts that are endian dependent.

Example:

```
#include<stdio.h>
int main()
{
    int *p;
    short q = 0x0123;

    p = (int *)&q;
    printf("%d\n", *p);
}
```

On a big-endian system, the output will be: 19070976. Whereas, on a little-endian system, the output will be: 291

Action:

Pointer type casting to different data size is unusual, and might be done by mistake - check if this is what you intended.

Reference:

4365 endian porting: the definition of the union may be endian dependent

Cause:

The values accessed using the members of a union may differ based on endianness of a machine in which the code is executed. This occurs when the union has members of different sizes, and when the value stored using one member is accessed using another of a different size.

Example:

```
#include <stdio.h>
union endian { short s; int i; char c;};
int main() {
    union endian u;
    u.i = 0x01234567;
    printf("%hx\n", u.s);
    return 0;
}
```

cadvice displays the following warning when the option `+Ww4365` is passed to it.

```
"/path/l.c", line 2: warning #4365-D: endian porting: the definition
of the union may be endian dependent
The field s is incompatible with field(s) i, c
The field i is incompatible with field(s) s, c
The field c is incompatible with field(s) s, i
union endian {
```

Action:

Avoid using unions to convert from one type to another of different size.

Reference:

4370 Control flows into the switch case from the previous case value

Cause:

Occurs when there is a fall-through case statement within a switch.

Example:

```
#include <stdio.h>
int foo(int a) {
    switch (a) {
        case 1: printf("One\n");
        case 2: printf("Two\n");
                break;
        case 3: printf("Three\n");
        case 4: printf("Four\n");
        default: printf("Bye\n");
    }
    return 0;
}
```

Action:

This warning is issued to ensure that the missing breaks between the different case values is expected.

Reference:

4372 Potential overflow in arithmetic expression involving time_t/clock_t values

Cause:

When arithmetic expressions that involve time_t/clock_t values overflow, the code might behave incorrectly.

Example:

```
#include <stdio.h>
```

```

int main() {
    int i;
    time_t t1 = 1024;
    scanf("%d", );
    if (t1+=i, t1 < 10) {
        t1 *= i;
    }
    return 0;
}

```

The expressions `t1 += i`, and `t1 *= i` can cause overflow and needs to be explicitly checked.

Action:

If there is no explicit check for overflow after arithmetic expressions on `time_t/clock_t` values, add a check.

Reference:

4373 non arithmetic integer conversion resulted in a change of sign

Cause:

When assigning hexadecimal or octal values to an integer type, if the data causes a change in the sign of the type, this warning is displayed.

Example:

```

int main()
{
    short i = 0xFFFF;
    short j = 0x10000;
    return 0;
}

```

Action:

Check for hexadecimal or octal constant, which though fitting in the integral type mentioned, causes a sign change.

Reference:

20035 variable %s is used before its value is set

Cause:

The compiler has detected use of a variable's value before it is set.

Example:

```

int func(int b)
{
    int a;
    a = a + b; // warning 20035 for use of variable a
    return a;
}

```

Action:

Initialize the reported variable with a value before it's use. This may not be the only use of the uninitialized value. So it is best to initialize variables as close as possible to the point of declaration.

Reference:

20036 variable %s (field %s) is used before its value is set

Cause:

The compiler has detected use of a member of a struct variable before it's value is set.

Example:

```
struct foo
{
    int a;
    int b;
};
int func()
{
    struct foo x;
    x.a = x.b + 10; // warning 20036 for use of field b
    return 0;
}
```

Action:

Initialize the struct member with a value before it's use. This may not be the only use of the uninitialized value. So it is best to initialize the field as close as possible to the point of struct variable declaration.

Reference:

20037 variable %s may be used before its value is set

Cause:

The compiler has detected use of a variable which might not always be used initialized before this use. One or more execution paths that lead to this use do not initialize this variable.

Example:

```
int func(int a)
{
    int b,c;
    if (a > 10)
        b = a;
    c = b + 10; // warning 20037 for use of variable b
    return c;
}
```

Action:

Check if all the execution paths leading to the usage of reported variable initialize it. To avoid such issues it is best to initialize the variable close to the point of declaration.

Reference:

20048 %s "%s" has incompatible type with previous declaration at line %s in file "%s"

Cause:

The declaration of a global variable or function does not match the corresponding declaration on a different file.

Example:

```
-- file1.c ----
typedef struct {
    int f;
    int g;
} A;
A global; // LINE 6
void print_A(A par) // LINE 8
{
    printf("%d %d", par.f, par.g);
}
-- file2.c ----
typedef struct {
    int f;
    char g; // Should be: int g;
```

```

} A;
A global;
void print_A(A);
main()
{
    print_A(global);
}
"file1.c", line 8: warning #20048-D: Function "print_A" has incompatible
type with previous declaration at line 8 in file "file2.c"
"file2.c", line 6: warning #20048-D: Variable "global" has incompatible
type with previous declaration at line 6 in file "file1.c"

```

Action:

In the case of mismatch for a global variable, review the 2 locations specified in the warning and make sure that the declared global types match. In the case of a function, review the 2 locations specified in the warning and verify that the types of the arguments and return value match for the two function declarations. This problem can usually be avoided by using header files to declare types or functions used across multiple source files.

Reference:

20072 variable %s is partially uninitialized when used

Cause:

An element in an array or a field of a struct is being referenced prior to its initialization.

Example:

```

int a[10];
    a[9] = 12;
    foo(a); // warning 20072: elements 0-8 are uninitialized

```

Action:

Ensure that all elements/fields of an array/struct are initialized prior to passing it as a function argument. .

Reference:

20073 variable %s may be partially uninitialized when used

Cause:

An element in an array or an unnamed field of a struct may be referenced prior to its initialization.

Example:

```

int a[2];
    a[0] = 10;
    if (cond)
        a[1] = 12;
    foo(a); // warning 20073: element 1 may be uninitialized

```

Action:

Ensure that all elements/fields of an array/struct are unconditionally initialized prior to accessing them.

Reference:

20074 variable %s (field "%s") may be used before its value is set

Cause:

A field of a struct may be referenced prior to its initialization.

Example:

```

struct A {
    int a;

```

```

        int b;
    };
    struct A a;
    a.a = 10;
    if (cond)
    a.b = 12;
    foo(a); // warning 20074: field b of struct a may
           // be uninitialized.

```

Action:

Ensure that all fields of a struct are unconditionally initialized prior to accessing them.

Reference:

20200 Potential null pointer dereference %s%s is detected %s

Cause:

A reference through a pointer whose value may be null has been detected. The pointer could have been conditionally initialized or assigned with the return value of a function that may return NULL.

Example:

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void nullptr3 ()
{
    char *p = malloc (20); // LINE 6
    strcpy (p, "nullptr3");
    FILE* f = fopen("blah","r"); // LINE 8
    if (p != NULL)
        fread(p,20,1,f);
    printf ("%s", p);
}
"file1.c", line 7, procedure nullptr3: warning #file1-D: Potential null
pointer dereference through p is detected (null definition:file1.c,
line 6)
"file1.c", line 10, procedure nullptr3: warning #file1-D: Potential null
pointer dereference through f is detected (null definition:file1.c,
line 8)

```

Action:

If you are assigning NULL to a pointer, make sure that no path in the program can dereference that pointer before it is assigned a different value. For routines that may return a NULL, check or assert that the value returned is not NULL. `char * pc = malloc(20); if (pc != NULL) strcpy(pc, "hello");` or: `char * pc = malloc(20); assert(pc != NULL); strcpy(pc, "hello");`

Reference:

20201 Memory leak is detected

Cause:

A memory leak is detected for the memory allocation done at the location mentioned in the message.

Example:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int leak1 (int k)
{
    char *p = malloc (k); // LINE 6
    strcpy (p, "hello");
    printf ("%s", p);
    return 0;
}
"memleak.c", line 6, procedure leak1: warning #20201-D: Memory
leak is detected.

```

Action:

Make sure that the pointer allocated at the location mentioned in the diagnostic is freed correctly.

Reference:

20202 Allocated memory may potentially be leaked %s

Cause:

A memory leak may occur for the memory allocation done at the location mentioned in the message. It is possible that the memory may be leaked along one of the paths from the allocation site.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int* leak2(int k, char* fname)
{
    FILE* f;
    int *p = (int*) malloc(k); // LINE 7
    if (p == 0)
        return 0;
    if (k > 10)
        return 0; // LINE 11
    return p;
}
"memleak.c", line 7, procedure leak2: warning #20202-D: Allocated memory
may potentially be leaked (at line 11)
```

Action:

Make sure that the pointer allocated at the location mentioned in the diagnostic, is freed correctly along all paths. For example, in the above case, if (k > 10), we return 0 without freeing p. After this point, p is unreachable from other pointers. Hence it is a potential memory leak along this point.

Reference:

20203 Potential out of scope use of local %s %s

Cause:

A local variable's address is being returned and dereferenced in the caller, or allocated memory is being returned and used in the caller or a variable defined in the inner scope is being accessed indirectly in the enclosing scope. Use of local variable outside its scope can lead to unexpected behavior.

Example:

```
#include<stdio.h>
int foo()
{
    int *p;
    {
        int q;
        scanf("%d", &q);
        p = &q;
    }
    // out of scope reference to q
    return *p;
}
int main()
{
    int result = foo();
    return result;
}
"oos.c", line 11, procedure foo: warning #20203-D: Potential out of scope use
of local variable q
```

Action:

Check that the local variable mentioned in the diagnostic does not have its address taken and returned to the caller function or an allocated memory pointer is not returned to the caller.

Reference:

20206 Out of bound access (%s)

Cause:

The expression is accessing out of object's memory boundary. The object could be an array, a heap memory buffer, or a string.

Example:

```
#include <string.h>
char buf[12];
struct A {
    int f1;
    int f2;
};
int i;
struct A a;
void foo()
{
    char c = 'd';
    i = *(int *)&c;           // LINE 12
    memset(buf, 0, 21);      // LINE 13
    memset(&a.f1, 0, sizeof(a)); // LINE 14
}
line 12, procedure foo: warning #20206-D: Out of bound access (In
expression "memset( (char*)buf, 0, 21 )", variable "buf" [j.c:2]
(type: char [12]) has byte range [0 .. 11], writing byte range [0
..20].)

line 13, procedure foo: warning #20206-D: Out of bound access (In
expression "* (int*)&c", variable "c" [j.c:11] (type: char ) has 1
byte, reading byte range [0 .. 3].)

line 14, procedure foo: warning #20206-D: Out of bound access (In
expression "memset( (char*)&(a)->f1, 97, 8 )", &(a)->f1 (type:
int) has byte range [0 .. 3], writing byte range [0 .. 7].)
```

Action:

Fix the out of bounds access if it is a real bug. Change the code to remove out of bound access if it is false positive. `i = c; memset(buf, 0, sizeof(buf); memset(&a, 0, sizeof(a));`

Reference:

20208 Forming out of bound address(%s)

Cause:

The expression is forming an address which is out of object's memory boundary.

Example:

```
char buf[12];
char *p;
void foo()
{
    int i = 20;
    p = &buf[0] + i;        // LINE 7
}
line 7, procedure foo: warning #20208-D: Forming out of bound
address (In expression "&buf[0]+20", variable "buf" [test.c:1]
(type: char [12]) has byte range [0 .. 11], forming address byte
[20].)
```

Action:

Fix the forming out of bounds access if it is a real bug.

Reference:

20210 Mismatch in allocation and deallocation

Cause:

The compiler has detected a code that uses different methods for memory allocation and its corresponding deallocation. A memory allocated using `malloc()` and its variations should be deallocated only by using `free()` and memory allocated using `new` operator should be deallocated only by using `delete`. Memory allocated using `alloca()` should not be deallocated using either `free()` or `delete`.

Example:

```
1 #include
2 #include
3
4 char* allocate()
5 {
6     return (char*)malloc(sizeof(char)*100);
7 }
8
9 void xyz()
10 {
11     char *str = allocate();
12     delete str;
13 }
```

```
"20210.C", line 12, procedure xyz: warning #20210-D: Mismatch in allocation
and deallocation
```

Action:

Replace the deallocator function call/operation with the one corresponding to the allocator or the vice versa.

Reference:

20213 Potential write to read only memory %s%s is detected

Cause:

The code is trying to modify an area of the memory that has been defined as read-only.

Example:

```
#include <stdlib.h>
#include <stdio.h>
const char * astring = "foo";
const char * bstring = "bar";
void mod_astring()
{
    char* w = (char*) astring;
    *w = 'g'; // LINE 9
}
int x;
void mod_bstring()
{
    char* s;
    if (x == 11)
        s = (char*) bstring;
    else
    {
        s = (char*) malloc(128);
        if (s == 0) exit(1);
    }
    sprintf(s,"%d",x); // LINE 25
    puts(s);
}
```

```
line 9, procedure mod_astring: warning #20213-D: Potential write to
readonly memory through astring is detected
```

```
line 25, procedure possible: warning #20213-D: Potential write to
readonly memory through s is detected
```

Action:

Look at the definition of the memory written to and make sure that it is not defined as "const".

Reference:

20229 variable "%s" is uninitialized if the loop %s is not executed

Cause:

This warning is issued by the compiler if a variable is initialized in the loop body. Initialization of this variable depends on the loop execution.

Example:

```
#include<stdio.h>

int foo(int n)
{
    int var;
    for (int i=0; i<n; i++) {
        var = i;
    }
    printf("var: %d\n", var);
    return var;
}

int main()
{
    foo(5);
    return 0;
}
```

Action:

Ensure that all the elements declared before the loop are not initialized inside the loop.

Reference:

20231 variable "%s" (field "%s") may be used before its value is set if the loop %s is not executed

Cause:

Example:

```
#include <stdio.h>
struct S { int v; int c;};

int foo2(int n)
{
    struct S var;
    for (int i=10; i<n; i++) {
        var.v = i;
    }
    var.c = 5;
    printf ("%d\n", var.v);
    return var.c;
}
int main()
{
    foo2(5);
    return 0;
}
```

Action:

Ensure that all elements or fields of an array or struct are unconditionally initialized prior to accessing them.

Reference: