1 Open Watcom C++ Diagnostic Messages

The following is a list of all warning and error messages produced by the Open Watcom C++ compilers. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%N`, `%S`, `%T`, `%s`, `%d` and `%u`. They represent strings that are substituted by the Open Watcom C++ compilers to make the error message more exact. `%d` and `%u` represent a string of digits; `%N`, `%S`, `%T` and `%s` a string, usually a symbolic name.

Consider the following program, named `err.cpp`, which contains errors.

Example:

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
File: err.cpp
(6,12): Error! E042: symbol 'i' already defined
  'i' declared at: (5,9)
(9,5): Error! E029: symbol 'x' has not been declared
err.cpp: 12 lines, included 174, no warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number and column of the line containing the error (in parentheses),
3. a message number, and
4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `err.cpp`. Error number 042 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `err.cpp`. Error number 029 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

A number of messages contain a reference to the ARM. This is the "Annotated C++ Reference Manual" written by Margaret A. Ellis and Bjarne Stroustrup and published by Addison-Wesley (ISBN 0-201-51459-1).

1.1 Diagnostic Messages

000 *internal compiler error*

If this message appears, please report the problem directly to the Open Watcom development team. See http://www.openwatcom.org/.

001 *assignment of constant found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

002 *constant out of range; truncated*

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

Example:

```
int a = 12345678901234567890;
```

003 *missing return value*

A function has been declared with a non-void return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

Example:

```
int foo( int a )
{
    int b = a + a;
}
```

The message will be issued at the end of the function.

004 *base class '%T' does not have a virtual destructor*

A virtual destructor has been declared in a class with base classes. However, one of those base classes does not have a virtual destructor. A ***delete*** of a pointer cast to such a base class will not function properly in all circumstances.

Example:

```
struct Base {
    ~Base();
};
struct Derived : Base {
    virtual ~Derived();
};
```

It is considered good programming practice to declare virtual destructors in all classes used as base classes of classes having virtual destructors.

005 *pointer or reference truncated*

The expression contains a transfer of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., assigning a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. This message indicates that some information is being lost so check the code carefully.

Example:

```
extern int __far *foo();
int __far *p_far = foo();
int __near *p_near = p_far; // truncated
```

006 *syntax error; probable cause: missing ';'*

The compiler has found a complete expression (or declaration) during parsing but could not continue. The compiler has detected that it could have continued if a semicolon was present so there may be a semicolon missing.

Example:

```
enum S {
}   // missing ';'

class X {
};
```

007 *'&array' may not produce intended result*

The type of the expression '&array' is different from the type of the expression 'array'. Suppose we have the declaration `char buffer[80].` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what one may have expected. The address-of operator '&' is not required for getting the address of an array.

008 *returning address of function argument or of auto or register variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

Example:

```
int *foo()
{
    int k = 123;
    return &k;      // k is automatic variable
}
```

009 *option requires a file name*

The specified option is not recognized by the compiler since there was no file name after it (i.e., "-fo=my.obj").

010 *asm directive ignored*

The asm directive (e.g., asm("mov r0,1");) is a non-portable construct. The Open Watcom C++ compiler treats all asm directives like comments.

011 *all members are private*

This message warns the programmer that there will be no way to use the contents of the class because all accesses will be flagged as erroneous (i.e., accessing a private member).

Example:

```
class Private {
    int a;
    Private();
    ~Private();
    Private( const Private& );
};
```

012 *template argument cannot be type '%T'*

A template argument can be either a generic type (e.g., `template < class` T >), a pointer, or an integral type. These types are required for expressions that can be checked at compile time.

013 *unreachable code*

The indicated statement will never be executed because there is no path through the program that causes control to reach that statement.

Example:

```
void foo( int *p )
{
    *p = 4;
    return;
    *p = 6;
}
```

The statement following the ***return*** statement cannot be reached.

014 *no reference to symbol '%S'*

There are no references to the declared variable. The declaration for the variable can be deleted. If the variable is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced),* or adding a statement that assigns the variable to itself.

015 *nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ISO/ANSI C. You may be missing the `*/` for the previous comment.

016 *template argument list cannot be empty*

An empty template argument list would result in a template that could only define a single class or function.

017 *label '%s' has not been referenced by a goto*

The indicated label has not been referenced and, as such, is useless. This warning can be safely ignored.

Example:

```
int foo( int a, int b )
{
un_refed:
    return a + b;
}
```

018 *no reference to anonymous union member '%S'*

The declaration for the anonymous member can be safely deleted without any effect.

019 *'break' may only appear in a for, do, while, or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace `{` for a ***while, do, for*** or ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    break;  // illegal
    return a+b;
}
```

020 *'case' may only appear in a switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    case 4:    // illegal
    return a+b;
}
```

021 *'continue' may only appear in a for, do, or while statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many } between the ***while, do*** or ***for*** statement and the ***continue*** statement.

Example:

```
int foo( int a, int b )
{
    continue;   // illegal
    return a+b;
}
```

022 *'default' may only appear in a switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many } between the start of the ***switch*** and the ***default*** label.

Example:

```
int foo( int a, int b )
{
    default: // illegal
    return a+b;
}
```

023 *misplaced '}' or missing earlier '{'*

An extra } has been found which cannot be matched up with an earlier {.

024 *misplaced #elif directive*

The ***#elif*** directive must be inside an ***#if*** preprocessing group and before the ***#else*** directive if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

025 *misplaced #else directive*

The ***#else*** directive must be inside an ***#if*** preprocessing group and follow all ***#elif*** directives if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

026 *misplaced #endif directive*

A ***#endif*** preprocessing directive has been found without a matching ***#if*** directive. You either have an extra ***#endif*** or you are missing an ***#if*** directive earlier in the file.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

027 *only one 'default' per switch statement is allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

Example:

```
int translate( int a )
{
    switch( a ) {
      case 1:
    a = 8;
    break;
      default:
    a = 9;
    break;
      default: // illegal
    a = 10;
    break;
    }
    return a;
}
```

028 *expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

029 *symbol '%N' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to ***#include*** a header file that contains the declaration.

Example:

```
int a = b;  // b has not been declared
```

030 *left expression must be a function or a function pointer*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

Example:

```
int a;
int b = a( 12 );
```

031 *operand must be an lvalue*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

Example:

```
void foo( int a )
{
    ( a + 1 ) = 7;
    int b = ++ ( a + 6 );
}
```

Both statements within the function are erroneous, since lvalues are expected where the additions are shown.

032 *label '%s' already defined*

All labels within a function must be unique.

Example:

```
void bar( int *p )
{
label:
    *p = 0;
label:
    return;
}
```

The second label is illegal.

033 *label '%s' is not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

Example:

```
void bar( int *p )
{
labl:
    *p = 0;
    goto label;
}
```

The label referenced in the ***goto*** is not defined.

034 *dimension cannot be zero*

The dimension of an array must be non-zero.

Example:

```
int array[0];   // not allowed
```

035 *dimension cannot be negative*

The dimension of an array must be positive.

Example:

```
int array[-1];  // not allowed
```

036 *dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

Example:

```
int array[][];   // not allowed
```

037 *invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

Example:

```
auto void foo()
{
}
```

038 *expression must have pointer type*

An attempt has been made to de-reference a variable or expression which is not declared to be a pointer.

Example:

```
int a;
int b = *a;
```

039 *cannot take address of an rvalue*

You can only take the address of a variable or memory location.

Example:

```
char c;
char *p1 = & & c;   // not allowed
char *p2 = & (c+1); // not allowed
```

040 *expression for '.' must be a class, struct or union*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***class, struct*** or ***union*** type.

Example:

```
struct S
{
    int a;
};
int &fun();
int a = fun().a;
```

041 *expression for '->' must be pointer to class, struct or union*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***class, struct*** or ***union*** type.

Example:

```
struct S
{
    int a;
};
int *fun();
int a = fun()->a;
```

042 *symbol '%S' already defined*

The specified symbol has already been defined.

Example:

```
char a = 2;
char a = 2; // not allowed
```

043 *static function '%S' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

Example:

```
static int fun( void );
int k = fun();
// fun not defined by end of program
```

044 *expecting label for goto statement*

The ***goto*** statement requires the name of a label.

Example:

```
int fun( void )
{
    goto;
}
```

045 *duplicate case value '%s' found*

Every case value in a ***switch*** statement must be unique.

Example:

```
int fun( int a )
{
    switch( a ) {
      case 1:
    return 7;
      case 2:
    return 9;
      case 1: // duplicate not allowed
    return 7;
    }
    return 79;
}
```

046 *bit-field width is too large*

The maximum field width allowed is 16 bits in the 16-bit compiler and 32 bits in the 32-bit compiler.

Example:

```
struct S
{
    unsigned bitfield :48;  // too wide
};
```

047 *width of a named bit-field must not be zero*

A bit field must be at least one bit in size.

Example:

```
struct S {
    int bitfield :10;
    int :0;   // okay, aligns to int
    int h :0; // error, field is named
};
```

048 *bit-field width must be positive*

You cannot have a negative field width.

Example:

```
struct S
{
    unsigned bitfield :-10; // cannot be negative
};
```

049 *bit-field base type must be an integral type*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

Example:

```
struct S
{
    float bitfield : 10;    // must be integral
};
```

050 *subscript on non-array*

One of the operands of '[]' must be an array or a pointer.

Example:

```
int array[10];
int i1 = array[0];  // ok
int i2 = 0[array];  // same as above
int i3 = 0[1];      // illegal
```

051 *incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

052 *argument for # must be a macro parm*

The argument for the stringize operator '#' must be a macro parameter.

053 *unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

Example:

```
#i_goofed   // not valid
```

054 *invalid #include directive*

A syntax error has been encountered in a ***#include*** directive.

Example:

```
#include    // no header file
#include stdio.h
```

Both examples are illegal.

055 *not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

Example:

```
#define mac(a,b) a+b
int i = mac(123);   // needs 2 parameters
```

056 *not expecting a return value*

The specified function is declared as a ***void*** function. Delete the ***return*** value, or change the type of the function.

Example:

```
void fun()
{
    return 14;  // not expecting return value
}
```

057 *cannot take address of a bit-field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

Example:

```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
void* p = &var.bitfield;    // illegal
```

058 *expression must be a constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

059 *unable to open '%s'*

The file specified in an ***#include*** directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the **INCLUDE** or **INCLUDE** environment variables or in the "i=" option on the command line.

060 *too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro. The extra parameters are ignored.

Example:

```
#define mac(a,b) a+b
int i = mac(1,2,3); // needs 2 parameters
```

061 *cannot use __based or __far16 pointers in this context*

The use of ***__based*** and ***__far16*** pointers is prohibited in ***throw*** expressions and ***catch*** statements.

Example:

```
extern int __based( __segname( "myseg" ) ) *pi;

void bad()
{
    try {
    throw pi;
    } catch( int __far16 *p16 ) {
    *p16 = 87;
    }
}
```

Both the ***throw*** expression and ***catch*** statements cause this error to be diagnosed.

062 *only one type is allowed in declaration specifiers*

Only one type is allowed for the first part of a declaration. A common cause of this message is that there may be a missing semi-colon (';') after a class definition.

Example:

```
class C
{
public:
    C();
}           // needs ";"

int foo() { return 7; }
```

063 *out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not ***#include*** header files that are not required.

064 *invalid character constant*

This message is issued for an improperly formed character constant.

Example:

```
char c = '12345';
char d = ''';
```

065 *taking address of variable with storage class 'register'*

You can take the address of a ***register*** variable in C++ (but not in ISO/ANSI C). If there is a chance that the source will be compiled using a C compiler, change the storage class from ***register*** to ***auto.***

Example:

```
extern int foo( char* );
int bar()
{
    register char c = 'c';
    return foo( &c );
}
```

066 *'delete' expression size is not allowed*

The C++ language has evolved to the point where the ***delete*** expression size is no longer required for a correct deletion of an array.

Example:

```
void fn( unsigned n, char *p ) {
    delete [n] p;
}
```

067 *ending " missing for string literal*

The compiler did not find a second double quote to end the string literal.

Example:

```
char *a = "no_ending_quote;
```

068 *invalid option*

The specified option is not recognized by the compiler.

069 *invalid optimization option*

The specified option is an unrecognized optimization option.

070 *invalid memory model*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

071 *expression must be integral*

An integral expression is required.

Example:

```
int foo( int a, float b, int *p )
{
    switch( a ) {
      case 1.3:     // must be integral
    return p[b];    // index not integer
      case 2:
    b <<= 2;    // can only shift integers
      default:
    return b;
    }
}
```

072 *expression must be arithmetic*

Arithmetic operations, such as "/" and "*", require arithmetic operands unless the operation has been overloaded or unless the operands can be converted to arithmetic operands.

Example:

```
class C
{
public:
    int c;
};
C cv;
int i = cv / 2;
```

073 *statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

Example:

```
extern int bar( int );
void foo( int a )
{
    if( a ) goto ending;
    bar( a );
ending:
    // needs statement following
}
```

074 *statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

075 *statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
int foo( int a )
{
    switch( a ) {
      default:
    return 7;
      case 1: // needs statement following
    }
    return 18;
}
```

076 *statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
int foo( int a )
{
    switch( a ) {
      case 7:
    return 7;
      default:
    // needs statement following
    }
    return 18;
}
```

077 *missing matching #endif directive*

You are missing a ***#endif*** to terminate a ***#if, #ifdef*** or ***#ifndef*** preprocessing directive.

Example:

```
#if 1
int a;
// needs #endif
```

078 *invalid macro definition, missing ')'*

The right parenthesis ")" is required for a function-like macro definition.

Example:

```
#define bad_mac( a, b
```

079 *missing ')' for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

Example:

```
#define mac( a, b) a+b
int d = mac( 1, 2
```

080 *%s*

This is a user message generated with the ***#error*** preprocessing directive.

Example:

```
#error my very own error message
```

081 *cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

Example:

```
typedef int TD(float);
TD array[12];
```

082 *function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

Example:

```
typedef int ARR[10];
ARR fun( float );
```

083 *function cannot return a function*

You cannot return a function. You can return a pointer to a function.

Example:

```
typedef int TD();
TD fun( float );
```

084 *function templates can only have type arguments*

A function template argument can only be a generic type (e.g., `template < class` T >). This is a restriction in the C++ language that allows compilers to automatically instantiate functions purely from the argument types of calls.

085 *maximum class size has been exceeded*

The 16-bit compiler limits the size of a ***struct*** or ***union*** to 64K so that the compiler can represent the offset of a member in a 16-bit register. This error also occurs if the size of a structure overflows the size of an ***unsigned*** integer.

Example:

```
struct S
{
    char arr1[ 0xfffe ];
    char arr2[ 0xfffe ];
    char arr3[ 0xfffe ];
    char arr4[ 0xfffffffe ];
};
```

086 *definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must ***#undef*** it before you can define it with a new definition.

Example:

```
#define CON 123
#define CON 124     // not same as previous
```

087 *initialization of '%S' must be in file scope*

A file scope variable must be initialized in file scope.

Example:

```
void fn()
{
    extern int v = 1;
}
```

088 *default argument for '%S' declared outside of class definition*

Problems can occur with member functions that do not declare all of their default arguments during the class definition. For instance, a copy constructor is declared if a class does not define a copy constructor. If a default argument is added later on to a constructor that makes it a copy constructor, an ambiguity results.

Example:

```
struct S {
    S( S const &, int );
    // S( S const & );  <-- declared by compiler
};
// ambiguity with compiler
// generated copy constructor
// S( S const & );
S::S( S const &, int = 0 )
{
}
```

089 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

Example:

```
#define badmac( a, b ) ## a ## b
```

090 *invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

Example:

```
float f = 123.9E+Q;
```

091 *'sizeof' is not allowed for a bit-field*

The smallest object that you can ask for the size of is a char.

Example:

```
struct S
{   int a;
    int b :10;
} v;
int k = sizeof( v.b );
```

092 *option requires a path*

The specified option is not recognized by the compiler since there was no path after it (i.e., "-i=d:\include;d:\path").

093 *must use 'va_start' macro inside function with variable arguments*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

Example:

```
#include <stdarg.h>
int foo( int a, int b )
{
    va_list args;
    va_start( args, a );
    va_end( args );
    return b;
}
```

094 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

095 *internal compiler error %d*

A bug has been encountered in the compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to the Open Watcom development team so that we can fix the problem. See http://www.openwatcom.org/.

096 *argument number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

097 *procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

098 *illegal register modified by '%s' #pragma*

For the 16-bit Open Watcom C/C++ compiler: The BP, CS, DS, and SS registers cannot be modified in small data models. The BP, CS, and SS registers cannot be modified in large data models.

For the 32-bit Open Watcom C/C++ compiler: The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data models. The EBP, CS, and SS registers cannot be modified in large data models.

099 *file must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function).

Note: This message has been disabled starting with Open Watcom v1.4. The ISO 1998 C++ standard allows empty translation units.

100 *out of macro space*

The compiler ran out of memory for storing macro definitions.

101 *keyboard interrupt detected*

The compilation has been aborted with Ctrl/C or Ctrl/Break.

102 *duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

Example:

```
    #define badmac( a, b, a ) a ## b
```

103 *unable to open work file: error code = %d*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

104 *write error on work file: error code = %d*

An error was encountered trying to write information to the work file. The disk could be full.

105 *read error on work file: error code = %d*

An error was encountered trying to read information from the work file.

106 *token too long; truncated*

The token must be less than 510 bytes in length.

107 *filename required on command line*

The name of a file to be compiled must be specified on the command line.

108 *command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the Open Watcom Compile and Link utility to compile multiple files with a single command.

109 *virtual member functions are not allowed in a union*

A union can only be used to overlay the storage of data. The storage of virtual function information (in a safe manner) cannot be done if storage is overlaid.

Example:

```
struct S1{ int f( int ); };
struct S2{ int f( int ); };
union un { S1 s1;
           S2 s2;
           virtual int vf( int );
         };
```

110 *union cannot be used as a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and achieve the same effect.

Example:

```
union U { int a; int b; };
class S : public U { int s; };
```

111 *union cannot have a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and inherit the base classes normally.

Example:

```
class S { public: int s; };
union U : public S { int a; int b; };
```

112 *cannot inherit an undefined base class '%T'*

The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:

```
class Undefined;
class C : public Undefined {
    int c;
};
```

113 *repeated direct base class will cause ambiguities*

Almost all accesses will be ambiguous. This restriction is useful in catching programming errors. The repeated base class can be encapsulated in another class if the repetition is required.

Example:

```
class Dup
{
    int d;
};
class C : public Dup, public Dup
{
    int c;
};
```

114 *templates may only be declared in namespace scope*

Currently, templates can only be declared in namespace scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

115 *linkages may only be declared in file scope*

A common source of errors for C and C++ result from the use of prototypes inside of functions. This restriction attempts to prevent such errors.

116 *unknown linkage '%s'*

Only the linkages "C" and "C++" are supported by Open Watcom C++.

Example:
```
extern "APL" void AplFunc( int* );
```

117 *too many storage class specifiers*

This message is a result of duplicating a previous storage class or having a different storage class. You can only have one of the following storage classes, ***extern, static, auto, register,*** or ***typedef.***

Example:
```
extern typedef int (*fn)( void );
```

118 *nameless declaration is not allowed*

A type was used in a declaration but no name was given.

Example:
```
static int;
```

119 *illegal combination of type specifiers*

An incorrect scalar type was found. Either a scalar keyword was repeated or the combination is illegal.

Example:
```
short short x;
short long y;
```

120 *illegal combination of type qualifiers*

A repetition of a type qualifier has been detected. Some compilers may ignore repetitions but strictly speaking it is incorrect code.

Example:
```
const const x;
struct S {
    int virtual virtual fn();
};
```

121 *syntax error*

The C++ compiler was unable to interpret the text starting at the location of the message. The C++ language is sufficiently complicated that it is difficult for a compiler to correct the error itself.

122 *parser stack corrupted*

The C++ parser has detected an internal problem that usually indicates a compiler problem. Please report this directly to the Open Watcom development team. See http://www.openwatcom.org/.

123 *template declarations cannot be nested within each other*

Currently, templates can only be declared in namespace scope. Furthermore, a template declaration must be finished before another template can be declared.

124 *expression is too complicated*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

125 *invalid redefinition of the typedef name '%S'*

Redefinition of typedef names is only allowed if you are redefining a typedef name to itself. Any other redefinition is illegal. You should delete the duplicate ***typedef*** definition.

Example:

```
typedef int TD;
typedef float TD;   // illegal
```

126 *class '%T' has already been defined*

This message usually results from the definition of two classes in the same scope. This is illegal regardless of whether the class definitions are identical.

Example:

```
class C {
};
class C {
};
```

127 *'sizeof' is not allowed for an undefined type*

If a type has not been defined, the compiler cannot know how large it is.

Example:

```
class C;
int x = sizeof( C );
```

128 *initializer for variable '%S' cannot be bypassed*

The variable may not be initialized when code is executing at the position indicated in the message. The C++ language places these restrictions to prevent the use of uninitialized variables.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 1:
    int b = 2;
    return b;
      default: // b bypassed
    return b + 5;
    }
}
```

129 *division by zero in a constant expression*

Division by zero is not allowed in a constant expression. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 4 / 0:  // illegal
    return a;
    }
    return a + 2;
}
```

130 *arithmetic overflow in a constant expression*

The multiplication of two integral values cannot be represented. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 0x7FFF * 0x7FFF * 0x7FFF:  // overflow
    return a;
    }
    return a + 2;
}
```

131 *not enough memory to fully optimize procedure '%s'*

The indicated procedure cannot be fully optimized with the amount of memory available. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

132 *not enough memory to maintain full peephole*

Certain optimizations benefit from being able to store the entire module in memory during optimization. All functions will be individually optimized but the optimizer will not be able to share code between functions if this message appears. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

133 *too many errors: compilation aborted*

The Open Watcom C++ compiler sets a limit to the number of error messages it will issue. Once the number of messages reaches the limit the above message is issued. This limit can be changed via the "/e" command line option.

134 *too many parm sets*

An extra parameter passing description has been found in the aux pragma text. Only one parameter passing description is allowed.

135 *'friend', 'virtual' or 'inline' modifiers may only be used on functions*

This message indicates that you are trying to declare a strange entity like an ***inline*** variable. These qualifiers can only be used on function declarations and definitions.

136 *more than one calling convention has been specified*

A function cannot have more than one #pragma modifier applied to it. Combine the pragmas into one pragma and apply it once.

137 *pure member function constant must be '0'*

The constant must be changed to '0' in order for the Open Watcom C++ compiler to accept the pure virtual member function declaration.

Example:

```
struct S {
    virtual int wrong( void ) = 91;
};
```

138 *based modifier has been repeated*

A repeated based modifier has been detected. There are no semantics for combining base modifiers so this is not allowed.

Example:

```
char *ptr;
char __based( void ) __based( ptr ) *a;
```

139 *enumeration variable is not assigned a constant from its enumeration*

In C++ (as opposed to C), enums represent values of distinct types. Thus, the compiler will not automatically convert an integer value to an enum type.

Example:

```
enum Days { sun, mod, tues, wed, thur, fri, sat };
enum Days day = 2;
```

140 *bit-field declaration cannot have a storage class specifier*

Bit-fields (along with most members) cannot have storage class specifiers in their declaration. Remove the storage class specifier to correct the code.

Example:

```
class C
{
public:
    extern unsigned bitf :10;
};
```

141 *bit-field declaration must have a base type specified*

A bit-field cannot make use of a default integer type. Specify the type ***int*** to correct the code.

Example:

```
class C
{
public:
    bitf :10;
};
```

142 *illegal qualification of a bit-field declaration*

A bit-field can only be declared ***const*** or ***volatile.*** Qualifications like ***friend*** are not allowed.

Example:

```
struct S {
    friend int bit1 :10;
    inline int bit2 :10;
    virtual int bit3 :10;
};
```

All three declarations of bit-fields are illegal.

143 *duplicate base qualifier*

The compiler has found a repetition of base qualifiers like ***protected*** or ***virtual.***

Example:

```
struct Base { int b; };
struct Derived : public public Base { int d; };
```

144 *only one access specifier is allowed*

The compiler has found more than one access specifier for a base class. Since the compiler cannot choose one over the other, remove the unwanted access specifier to correct the code.

Example:

```
struct Base { int b; };
struct Derived : public protected Base { int d; };
```

145 *unexpected type qualifier found*

Type specifiers cannot have ***const*** or ***volatile*** qualifiers. This shows up in ***new*** expressions because one cannot allocate a ***const*** object.

146 *unexpected storage class specifier found*

Type specifiers cannot have ***auto*** or ***static*** storage class specifiers. This shows up in ***new*** expressions because one cannot allocate a ***static*** object.

147 *access to '%S' is not allowed because it is ambiguous*

There are two ways that this error can show up in C++ code. The first way a member can be ambiguous is that the same name can be used in two different classes. If these classes are combined with multiple inheritance, accesses of the name will be ambiguous.

Example:

```
struct S1 { int s; };
struct S2 { int s; };
struct Der : public S1, public S2
{
    void foo() { s = 2; };  // s is ambiguous
};
```

The second way a member can be ambiguous involves multiple inheritance. If a class is inherited non-virtually by two different classes which then get combined with multiple inheritance, an access of the member is faced with deciding which copy of the member is intended. Use the '::' operator to clarify what member is being accessed or access the member with a different class pointer or reference.

Example:

```
struct Top { int t; };
struct Mid : public Top { int m; };
struct Bot : public Top, public Mid
{
    void foo() { t = 2; };  // t is ambiguous
};
```

148 *access to private member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access private members of the class.

Example:

```
struct Top { int t; };
class Bot : private Top
{
    int foo() { return t; };  // t is private
};
Bot b;
int k = b.foo();    // foo is private
```

149 *access to protected member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access protected members of the class. The compiler also requires that ***protected*** members be accessed through a derived class to ensure that an unrelated base class cannot be quietly modified. This is a fairly recent change to the C++ language that may cause Open Watcom C++ to not accept older C++ code. See Section 11.5 in the ARM for a discussion of protected access.

Example:

```
struct Top { int t; };
struct Mid : public Top { int m; };
class Bot : protected Mid
{
protected:
    // t cannot be accessed
    int foo() { return t; };
};
Bot b;
int k = b.foo(); // foo is protected
```

150 *operation does not allow both operands to be pointers*

There may be a missing indirection in the code exhibiting this error. An example of this error is adding two pointers.

Example:

```
void fn()
{
    char *p, *q;

    p += q;
}
```

151 *operand is neither a pointer nor an arithmetic type*

An example of this error is incrementing a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    ++x;
}
```

152 *left operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    x = x + 1;
}
```

153 *right operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    x = 1 + x;
}
```

154 *cannot subtract a pointer from an arithmetic operand*

The subtract operands are probably in the wrong order.

Example:

```
int fn( char *p )
{
    return( 10 - p );
}
```

155 *left expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = x * 1;
}
```

156 *right expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = 1 * x;
}
```

157 *left expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:

```
struct S { } x;
void fn()
{
    x = x ^ 1;
}
```

158 *right expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:

```
struct S { } x;
void fn()
{
    x = 1 ^ x;
}
```

159 *cannot assign a pointer value to an arithmetic item*

The pointer value must be cast to the desired type before the assignment takes place.

Example:

```
void fn( char *p )
{
    int a;

    a = p;
}
```

160 *attempt to destroy a far object when the data model is near*

Destructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    ~Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

161 *attempt to call member function for far object when the data model is near*

Member functions cannot be called for objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    int foo();
};

Obj far obj;
int integer = obj.foo();
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

162 *template type argument cannot have a default argument*

This message was produced by earlier versions of the Open Watcom C++ compiler. Support for default template arguments was added in version 1.3 and this message was removed at that time.

163 *attempt to delete a far object when the data model is near*

delete cannot be used to deallocate objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
};

void foo( Obj far *p )
{
    delete p;
}
```

The second last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

164 *first operand is not a class, struct or union*

The ***offsetof*** operation can only be performed on a type that can have members. It is meaningless for any other type.

Example:

```
#include <stddef.h>

int fn( void )
{
    return offsetof( double, sign );
}
```

165 *syntax error: class template cannot be processed*

The class template contains unbalanced braces. The class definition cannot be processed in this form.

166 *cannot convert right pointer to type of left operand*

The C++ language will not allow the implicit conversion of unrelated class pointers. An explicit cast is required.

Example:

```
class C1;
class C2;

void fun( C1* pc1, C2* pc2 )
{
    pc2 = pc1;
}
```

167 *left operand must be an lvalue*

The left operand must be an expression that is valid on the left side of an assignment. Examples of incorrect lvalues include constants and the results of most operators.

Example:

```
int i, j;
void fn()
{
    ( i - 1 ) = j;
    1 = j;
}
```

168 *static data members are not allowed in an union*

A union should only be used to organize memory in C++. Enclose the union in a class if you need a static data member associated with the union.

Example:

```
union U
{
    static int a;
    int b;
    int c;
};
```

169 *invalid storage class for a member*

A class member cannot be declared with ***auto, register,*** or ***extern*** storage class.

Example:

```
class C
{
    auto int a;     // cannot specify auto
};
```

170 *declaration is too complicated*

The declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

Example:

```
int ************p;
```

Example:

```
// transform this to ...
typedef int ****PD1;
typedef PD1 ****PD2;
PD2 ****p;
```

171 *exception declaration is too complicated*

The exception declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

172 *floating-point constant too large to represent*

The Open Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the positive exponent is too large.

Example:

```
float f = 1.2e78965;
```

173 *floating-point constant too small to represent*

The Open Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the negative exponent is too large.

Example:

```
float f = 1.2e-78965;
```

174 *class template '%M' cannot be overloaded*

A class template name must be unique across the entire C++ program. Furthermore, a class template cannot coexist with another class template of the same name.

175 *range of enum constants cannot be represented*

If one integral type cannot be chosen to represent all values of an enumeration, the values cannot be used reliably in the generated code. Shrink the range of enumerator values used in the ***enum*** declaration.

Example:

```
enum E
{   e1 = 0xFFFFFFFF
,   e2 = -1
};
```

176 *'%S' cannot be in the same scope as a class template*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

177 *invalid storage class in file scope*

A declaration in file scope cannot have a storage class of ***auto*** or ***register.***

Example:

```
auto int a;
```

178 *const object must be initialized*

Constant objects cannot be modified so they must be initialized before use.

Example:

```
const int a;
```

179 *declaration cannot be in the same scope as class template '%S'*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

180 *template arguments must be named*

A member function of a template class cannot be defined outside the class declaration unless all template arguments have been named.

181 *class template '%M' is already defined*

A class template cannot have its definition repeated regardless of whether it is identical to the previous definition.

182 *invalid storage class for an argument*

An argument declaration cannot have a storage class of ***extern, static,*** or ***typedef.***

Example:

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

183 *unions cannot have members with constructors*

A union should only be used to organize memory in C++. Allowing union members to have constructors would mean that the same piece of memory could be constructed twice.

Example:

```
class C
{
    C();
};
union U
{
    int a;
    C c;    // has constructor
};
```

184 *statement is too complicated*

The statement contains too many nested constructs. Break up the statement into multiple statements.

185 *'%s' is not the name of a class or namespace*

The right hand operand of a '::' operator turned out not to reference a class type or namespace. Because the name is followed by another '::', it must name a class or namespace.

186 *attempt to modify a constant value*

Modification of a constant value is not allowed. If you must force this to work, take the address and cast away the constant nature of the type.

Example:

```
static int const con = 12;
void foo()
{
    con = 13;       // error
    *(int*)&con = 13;   // ok
}
```

187 *'offsetof' is not allowed for a bit-field*

A bit-field cannot have a simple offset so it cannot be referenced in an ***offsetof*** expression.

Example:

```
#include <stddef.h>
struct S
{
    unsigned b1 :10;
    unsigned b2 :15;
    unsigned b3 :11;
};
int k = offsetof( S, b2 );
```

188 *base class is inherited with private access*

This warning indicates that the base class was originally declared as a ***class*** as opposed to a ***struct.*** Furthermore, no access was specified so the base class defaults to ***private*** inheritance. Add the ***private*** or ***public*** access specifier to prevent this message depending on the intended access.

189 *overloaded function cannot be selected for arguments used in call*

Either conversions were not possible for an argument to the function or a function with the right number of arguments was not available.

Example:

```
class C1;
class C2;
int foo( C1* );
int foo( C2* );
int k = foo( 5 );
```

190 *base operator operands must be " __segment :> pointer "*

The base operator (:>) requires the left operand to be of type __segment and the right operand to be a pointer.

Example:

```
char _based( void ) *pcb;
char __far *pcf = pcb;      // needs :> operator
```

Examples of typical uses are as follows:

Example:

```
const __segment mySegAbs = 0x4000;
char __based( void ) *c_bv = 24;
char __far *c_fp_1 = mySegAbs :> c_bv;
char __far *c_fp_2 = __segname( "_DATA" ) :> c_bv;
```

191 *expression must be a pointer or a zero constant*

In a conditional expression, if one side of the ':' is a pointer then the other side must also be a pointer or a zero constant.

Example:

```
extern int a;
int *p = ( a > 7 ) ? &a : 12;
```

192 *left expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = p + 2;
```

193 *right expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = 2 + p;
```

194 *expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = ++p;
```

195 *'sizeof' is not allowed for a function*

A function has no size defined for it by the C++ language specification.

Example:

```
typedef int FT( int );

unsigned y = sizeof( FT );
```

196 *'sizeof' is not allowed for type void*

The type ***void*** has no size defined for it by the C++ language specification.

Example:

```
void *p;
unsigned size = sizeof( *p );
```

197 *type cannot be defined in this context*

A type cannot be defined in certain contexts. For example, a new type cannot be defined in an argument list, a ***new*** expression, a conversion function identifier, or a catch handler.

Example:

```
extern int goop();
int foo()
{
    try {
    return goop();
    } catch( struct S { int s; } ) {
    return 2;
    }
}
```

198 *expression cannot be used as a class template parameter*

The compiler has to be able to compare expressions during compilation so this limits the complexity of expressions that can be used for template parameters. The only types of expressions that can be used for template parameters are constant integral expressions and addresses. Any symbols must have external linkage or must be static class members.

199 *premature end-of-file encountered during compilation*

The compiler expects more source code at this point. This can be due to missing parentheses (')') or missing closing braces ('}').

200 *duplicate case value '%s' after conversion to type of switch expression*

A duplicate ***case*** value has been found. Keep in mind that all case values must be converted to the type of the switch expression. Constants that may be different initially may convert to the same value.

Example:

```
enum E { e1, e2 };
void foo( short a )
{
    switch( a ) {
      case 1:
      case 0x10001:    // converts to 1 as short
    break;
    }
}
```

201 *declaration statement follows an if statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    if( a )
    int b = 14;
}
```

202 *declaration statement follows an else statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    if( a )
    int c = 15;
    else
    int b = 14;
}
```

203 *declaration statement follows a switch statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    switch( a )
    int b = 14;
}
```

204 *'this' pointer is not defined*

The ***this*** value can only be used from within non-static member functions.

Example:

```
void *fn()
{
    return this;
}
```

205 *declaration statement cannot follow a while statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    while( a )
    int b = 14;
}
```

206 *declaration statement cannot follow a do statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    do
    int b = 14;
    while( a );
}
```

207 *declaration statement cannot follow a for statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended. A ***for*** loop with an initial declaration is allowed to be used within another ***for*** loop, so this code is legal C++:

Example:

```
void fn( int **a )
{
    for( int i = 0; i < 10; ++i )
    for( int j = 0; j < 10; ++j )
        a[i][j] = i + j;
}
```

The following example, however, illustrates a potentially erroneous situation.

Example:

```
void foo( int a )
{
    for( ; a<10; )
    int b = 14;
}
```

208 *pointer to virtual base class converted to pointer to derived class*

Since the relative position of a virtual base can change through repeated derivations, this conversion is very dangerous. All C++ translators must report an error for this type of conversion.

Example:

```
struct VBase { int v; };
struct Der : virtual public VBase { int   d; };
extern VBase *pv;
Der *pd = (Der *)pv;
```

209 *cannot use far pointer in this context*

Only near pointers can be thrown when the data memory model is near.

Example:

```
extern int __far *p;
void foo()
{
    throw p;
}
```

When the small memory model (-ms switch) is selected, the ***throw*** expression is diagnosed as erroneous. Similarly, only near pointers can be specified in ***catch*** statements when the data memory model is near.

210 *returning reference to function argument or to auto or register variable*

The storage for the automatic variable will be destroyed immediately upon function return. Returning a reference effectively allows the caller to modify storage which does not exist.

Example:

```
class C
{
    char *p;
public:
    C();
    ~C();
};

C& foo()
{
    C auto_var;
    return auto_var;    // not allowed
}
```

211 *#pragma attributes for '%S' may be inconsistent*

A pragma attribute was changed to a value which matches neither the current default not the previous value for that attribute. A warning is issued since this usually indicates an attribute is being set twice (or more) in an inconsistent way. The warning can also occur when the default attribute is changed between two pragmas for the same object.

212 *function arguments cannot be of type void*

Having more than one ***void*** argument is not allowed. The special case of one ***void*** argument indicates that the function accepts no parameters.

Example:

```
void fn1( void )        // OK
{
}
void fn2( void, void, void )    // Error!
{
}
```

213 *class template '%M' requires more parameters for instantiation*

The class template instantiation has too few parameters supplied so the class cannot be instantiated properly.

214 *class template '%M' requires fewer parameters for instantiation*

The class template instantiation has too many parameters supplied so the class cannot be instantiated properly.

215 *no declared 'operator new' has arguments that match*

An ***operator new*** could not be found to match the ***new*** expression. Supply the correct arguments for special ***operator new*** functions that are defined with the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void *operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

216 *wide character string concatenated with a simple character string*

There are no semantics defined for combining a wide character string with a simple character string. To correct the problem, make the simple character string a wide character string by prefixing it with a ***L.***

Example:

```
char *p = "1234" L"5678";
```

217 *'offsetof' is not allowed for a static member*

A ***static*** member does not have an offset like simple data members. If this is required, use the address of the ***static*** member.

Example:

```
#include <stddef.h>
class C
{
public:
    static int stat;
    int memb;
};

int size_1 = offsetof( C, stat );   // not allowed
int size_2 = offsetof( C, memb );   // ok
```

218 *cannot define an array of void*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot declare an array of ***void.***

Example:

```
void array[24];
```

219 *cannot define an array of references*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating an array of references is currently not allowed in the C++ language.

Example:

```
int& array[24];
```

220 *cannot define a reference to void*

One cannot create a reference to a ***void*** because there can be no ***void*** variables to supply for initializing the reference.

Example:

```
void& ref;
```

221 *cannot define a reference to another reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a reference to another reference is currently not allowed in the C++ language.

Example:

```
int & & ref;
```

222 *cannot define a pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a pointer to a reference is currently not allowed in the C++ language.

Example:

```
char& *ptr;
```

223 *cannot initialize array with 'operator new'*

The initialization of arrays created with ***operator new*** can only be done with default constructors. The capability of using another constructor with arguments is currently not allowed in the C++ language.

Example:

```
struct S
{
    S( int );
};
S *p = new S[10] ( 12 );
```

224 *'%N' is a variable of type void*

A variable cannot be of type ***void.*** The ***void*** type can only be used in restricted circumstances because it has no size. For instance, a function returning ***void*** means that it does not return any value. A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

225 *cannot define a member pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a member pointer to a reference is currently not allowed in the C++ language.

Example:

```
struct S
{
    S();
    int &ref;
};

int& S::* p;
```

226 *function '%S' is not distinct*

The function being declared is not distinct enough from the other functions of the same name. This means that all function overloads involving the function's argument types will be ambiguous.

Example:

```
struct S {
    int s;
};
extern int foo( S* );
extern int foo( S* const ); // not distinct enough
```

227 *overloaded function is ambiguous for arguments used in call*

The compiler could not find an unambiguous choice for the function being called.

Example:

```
extern int foo( char );
extern int foo( short );
int k = foo( 4 );
```

228 *declared 'operator new' is ambiguous for arguments used*

The compiler could not find an unambiguous choice for ***operator new.***

Example:

```
#include <stdlib.h>
struct Der
{
    int s[2];
    void* operator new( size_t, char );
    void* operator new( size_t, short );
};
Der *p = new(10) Der;
```

229 *function '%S' has already been defined*

The function being defined has already been defined elsewhere. Even if the two function bodies are identical, there must be only one definition for a particular function.

Example:

```
int foo( int s ) { return s; }
int foo( int s ) { return s; }  // illegal
```

230 *expression on left is an array*

The array expression is being used in a context where only pointers are allowed.

Example:

```
void fn( void *p )
{
    int a[10];

    a = 0;
    a = p;
    a++;
}
```

231 *user-defined conversion has a return type*

A user-defined conversion cannot be declared with a return type. The "return type" of the user-defined conversion is implicit in the name of the user-defined conversion.

Example:

```
struct S {
    int operator int(); // cannot have return type
};
```

232 *user-defined conversion must be a function*

The operator name describing a user-defined conversion can only be used to designate functions.

Example:

```
// operator char can only be a function
int operator char = 9;
```

233 *user-defined conversion has an argument list*

A user-defined conversion cannot have an argument list. Since user-defined conversions can only be non-static member functions, they have an implicit ***this*** argument.

Example:

```
struct S {
    operator int( S& ); // cannot have arguments
};
```

234 *destructor cannot have a return type*

A destructor cannot have a return type (even ***void***). The destructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:

```
struct S {
    void* ~S();
};
```

235 *destructor must be a function*

The tilde ('~') style of name is reserved for declaring destructor functions. Variable names cannot make use of the destructor style of names.

Example:

```
struct S {
    int ~S; // illegal
};
```

236 *destructor has an argument list*

A destructor cannot have an argument list. Since destructors can only be non-static member functions, they have an implicit ***this*** argument.

Example:

```
struct S {
    ~S( S& );
};
```

237 *'%N' must be a function*

The ***operator*** style of name is reserved for declaring operator functions. Variable names cannot make use of the ***operator*** style of names.

Example:

```
struct S {
    int operator+;  // illegal
};
```

238 *'%N' is not a function*

The compiler has detected what looks like a function body. The message is a result of not finding a function being declared. This can happen in many ways, such as dropping the ':' before defining base classes, or dropping the '=' before initializing a structure via a braced initializer.

Example:

```
struct D B { int i; };
```

239 *nested type class '%s' has not been declared*

A nested class has not been found but is required by the use of repeated '::' operators. The construct "A::B::C" requires that 'A' be a class type, and 'B' be a nested class within the scope of 'A'.

Example:

```
struct B {
    static int b;
};
struct A : public B {
};
int A::B::b = 2;    // B not nested in A
```

The preceding example is illegal; the following is legal

Example:

```
struct A {
    struct B {
    static int b;
    };
};
int A::B::b = 2;    // B nested in A
```

240 *enum '%s' has not been declared*

An elaborated reference to an ***enum*** could not be satisfied. All enclosing scopes have been searched for an ***enum*** name. Visible variable declarations do not affect the search.

Example:

```
struct D {
    int i;
    enum E { e1, e2, e3 };
};
enum E enum_var;    // E not visible
```

241 *class or namespace '%s' has not been declared*

The construct "A::B::C" requires that 'A' be a class type or a namespace, and 'B' be a nested class or namespace within the scope of 'A'. The reference to 'A' could not be satisfied. All enclosing scopes have been searched for a ***class*** or ***namespace*** name. Visible variable declarations do not affect the search.

Example:

```
struct A{ int a; };

int b;
int c = B::A::b;
```

242 *only one initializer argument allowed*

The comma (',') in a function like cast is treated like an argument list comma (','). If a comma expression is desired, use parentheses to enclose the comma expression.

Example:

```
void fn()
{
    int a;

    a = int( 1, 2 );        // Error!
    a = int( ( 1, 2 ) );    // OK
}
```

243 *default arguments are not part of a function's type*

This message indicates that a declaration has been found that requires default arguments to be part of a function's type. Either declaring a function ***typedef*** or a pointer to a function with default arguments are examples of incorrect declarations.

Example:

```
typedef int TD( int, int a = 14 );
int (*p)( int, int a = 14 ) = 0;
```

244 *missing default arguments*

Gaps in a succession of default arguments are not allowed in the C++ language.

Example:

```
void fn( int = 1, int, int = 3 );
```

245 *overloaded operator cannot have default arguments*

Preventing overloaded operators from having default arguments enforces the property that binary operators will only be called from a use of a binary operator. Allowing default arguments would allow a binary ***operator +*** to function as a unary ***operator +.***

Example:

```
class C
{
public:
    C operator +( int a = 10 );
};
```

246 *left expression is not a pointer to a constant object*

One cannot assign a pointer to a constant type to a pointer to a non-constant type. This would allow a constant object to be modified via the non-constant pointer. Use a cast if this is absolutely necessary.

Example:

```
char* fun( const char* p )
{
    char* q;
    q = p;
    return q;
}
```

247 *cannot redefine default argument for '%S'*

Default arguments can only be defined once in a program regardless of whether the value of the default argument is identical.

Example:

```
static int foo( int a = 10 );
static int foo( int a = 10 )
{
    return a+a;
}
```

248 *using default arguments would be overload ambiguous with '%S'*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name.

Example:

```
void fn( int );
void fn( int, int = 1 );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' without any default arguments or the second 'fn' with a default argument applied.

249 *using default arguments would be overload ambiguous with '%S' using default arguments*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name with default arguments.

Example:

```
void fn( int, int = 1 );
void fn( int, char = 'a' );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' with a default argument or the second 'fn' with a default argument applied.

250 *missing default argument for '%S'*

In C++, one is allowed to add default arguments to the right hand arguments of a function declaration in successive declarations. The message indicates that the declaration is only valid if there was a default argument previously declared for the next argument.

Example:

```
void fn1( int    , int     );
void fn1( int    , int = 3 );
void fn1( int = 2, int     );   // OK

void fn2( int    , int     );
void fn2( int = 2, int     );   // Error!
```

251 *enum references must have an identifier*

There is no way to reference an anonymous ***enum.*** If all enums are named, the cause of this message is most likely a missing identifier.

Example:

```
enum   { X, Y, Z }; // anonymous enum
void fn()
{
    enum *p;
}
```

252 *class declaration has not been seen for '~%s'*

A destructor has been used in a context where its class is not visible.

Example:

```
class C;

void fun( C* p )
{
    p->~S();
}
```

253 *'::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** member functions. The Open Watcom C++ compiler also allows code that is qualified with its own class so that declarations can be moved in and out of class definitions easily.

Example:

```
struct N {
    void bar();
};
struct S {
    void S::foo() { // OK
    }
    void N::bar() { // error
    }
};
```

254 *'%S' has not been declared as a member*

In a definition of a class member, the indicated declaration must already have been declared when the class was defined.

Example:

```
class C
{
public:
    int c;
    int goop();
};
int C::x = 1;
C::not_decled() { }
```

255 *default argument expression cannot use function argument '%S'*

Default arguments must be evaluated at each call. Since the order of evaluation for arguments is undefined, a compiler must diagnose all default arguments that depend on other arguments.

Example:

```
void goop( int d )
{
    struct S {
    // cannot access "d"
    int foo( int c, int b = d )
        {
        return b + c;
        };
    };
}
```

256 *default argument expression cannot use local variable '%S'*

Default arguments must be evaluated at each call. Since a local variable is not always available in all contexts (e.g., file scope initializers), a compiler must diagnose all default arguments that depend on local variables.

Example:

```
void goop( void )
{
    int a;
    struct S {
    // cannot access "a"
    int foo( int c, int b = a )
        {
        return b + c;
        };
    };
}
```

257 *access declarations may only be 'public' or 'protected'*

Access declarations are used to increase access. A ***private*** access declaration is useless because there is no access level for which ***private*** is an increase in access.

Example:

```
class Base
{
    int pri;
protected:
    int pro;
public:
    int pub;
};
class Derived : public Base
{
    private: Base::pri;
};
```

258 *cannot declare both a function and variable of the same name ('%N')*

Functions can be overloaded in C++ but they cannot be overloaded in the presence of a variable of the same name. Likewise, one cannot declare a variable in the same scope as a set of overloaded functions of the same name.

Example:

```
int foo();
int foo;
struct S {
    int bad();
    int bad;
};
```

259 *class in access declaration ('%T') must be a direct base class*

Access declarations can only be applied to direct (immediate) base classes.

Example:

```
struct B {
    int f;
};
struct C : B {
    int g;
};
struct D : private C {
    B::f;
};
```

In the above example, "C" is a direct base class of "D" and "B" is a direct base class of "C", but "B" is not a direct base class of "D".

260 *overloaded functions ('%N') do not have the same access*

If an access declaration is referencing a set of overloaded functions, then they all must have the same access. This is due to the lack of a type in an access declaration.

Example:

```
class C
{
    static int foo( int );     // private
public:
    static int foo( float );   // public
};

class B : private C
{
public: C::foo;
};
```

261 *cannot grant access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : private Base
{
    public: Base::pub;       // ok
    public: Base::pro;       // changes access
};
```

262 *cannot reduce access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : public Base
{
    protected: Base::pub;   // changes access
    protected: Base::pro;   // ok
};
```

263 *nested class '%N' has not been defined*

The current state of the C++ language supports nested types. Unfortunately, this means that some working C code will not work unchanged.

Example:

```
struct S {
    struct T;
    T *link;
};
```

In the above example, the class "T" will be reported as not being defined by the end of the class declaration. The code can be corrected in the following manner.

Example:

```
struct S {
    struct T;
    T *link;
    struct T {
    };
};
```

264 *user-defined conversion must be a non-static member function*

A user-defined conversion is a special member function that allows the class to be converted implicitly (or explicitly) to an arbitrary type. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:

```
struct S
{
    static operator int();
};
```

265 *destructor must be a non-static member function*

A destructor is a special member function that will perform cleanup on a class before the storage for the class will be released. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:

```
struct S
{
    static ~S();
};
```

266 *'%N' must be a non-static member function*

The operator function in the message is restricted to being a non-static member function. This usually means that the operator function is treated in a special manner by the compiler.

Example:

```
class C
{
public:
    static operator =( C&, int );
};
```

267 *'%N' must have one argument*

The operator function in the message is only allowed to have one argument. An operator like ***operator ~*** is one such example because it represents a unary operator.

Example:

```
class C
{
public: int c;
};
C& operator~( const C&, int );
```

268 *'%N' must have two arguments*

The operator function in the message must have two arguments. An operator like ***operator +=*** is one such example because it represents a binary operator.

Example:

```
class C
{
public: int c;
};
C& operator += ( const C& );
```

269 *'%N' must have either one argument or two arguments*

The operator function in the message must have either one argument or two arguments. An operator like ***operator +*** is one such example because it represents either a unary or a binary operator.

Example:

```
class C
{
public: int c;
};
C& operator+( const C&, int, float );
```

270 *'%N' must have at least one argument*

The ***operator new*** and ***operator new []*** member functions must have at least one argument for the size of the allocation. After that, any arguments are up to the programmer. The extra arguments can be supplied in a ***new*** expression via the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void * operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

271 *'%N' must have a return type of void*

The C++ language requires that ***operator delete*** and ***operator delete []*** have a return type of ***void.***

Example:

```
class C
{
public:
    int c;
    C* operator delete( void* );
    C* operator delete []( void* );
};
```

272 *'%N' must have a return type of pointer to void*

The C++ language requires that both ***operator new*** and ***operator new []*** have a return type of `void *`.

Example:

```
#include <stddef.h>
class C
{
public:
    int c;
    C* operator new( size_t size );
    C* operator new []( size_t size );
};
```

273 *the first argument of '%N' must be of type size_t*

The C++ language requires that the first argument for ***operator new*** and ***operator new []*** be of the type "size_t". The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:

```
void *operator new( int size );
void *operator new( double size, char c );
void *operator new []( int size );
void *operator new []( double size, char c );
```

274 *the first argument of '%N' must be of type pointer to void*

The C++ language requires that the first argument for ***operator delete*** and ***operator delete []*** be a `void *`.

Example:

```
class C;
void operator delete( C* );
void operator delete []( C* );
```

275 *the second argument of '%N' must be of type size_t*

The C++ language requires that the second argument for ***operator delete*** and ***operator delete []*** be of type "size_t". The two argument form of ***operator delete*** and ***operator delete []*** is optional and it can only be present inside of a class declaration. The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:

```
struct S {
    void operator delete( void *, char );
    void operator delete []( void *, char );
};
```

276 *the second argument of 'operator ++' or 'operator --' must be int*

The C++ language requires that the second argument for ***operator ++*** be ***int.*** The two argument form of ***operator ++*** is used to overload the postfix operator "++". The postfix operator "--" can be overloaded similarly.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( C&, unsigned );
```

277 *return type of '%S' must allow the '->' operator to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.***

Example:

```
struct S {
    int a;
    S *operator ->();
};

void fn( S &q )
{
    q->a = 1; // becomes (q.operator ->())->a = 1;
}
```

278 *'%N' must take at least one argument of a class/enum or a reference to a class/enum*

Overloaded operators can only be defined for classes and enumerations. At least one argument, must be a class or an enum type in order for the C++ compiler to distinguish the operator from the built-in operators.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( unsigned, int );
```

279 *too many initializers*

The compiler has detected extra initializers.

Example:

```
int a[3] = { 1, 2, 3, 4 };
```

280 *too many initializers for character string*

A string literal used in an initialization of a character array is viewed as providing the terminating null character. If the number of array elements isn't enough to accept the terminating character, this message is output.

Example:

```
char ac[3] = "abc";
```

281 *expecting '%s' but found expression*

This message is output when some bracing or punctuation is expected but an expression was encountered.

Example:

```
int b[3] = 3;
```

282 *anonymous struct/union member '%N' cannot be declared in this class*

An anonymous member cannot be declared with the same name as its containing class.

Example:

```
struct S {
    union {
    int S;      // Error!
    char b;
    };
};
```

283 *unexpected '%s' during initialization*

This message is output when some unexpected bracing or punctuation is encountered during initialization.

Example:

```
int e = { { 1 };
```

284 *nested type '%N' cannot be declared in this class*

A nested type cannot be declared with the same name as its containing class.

Example:

```
struct S {
    typedef int S;  // Error!
};
```

285 *enumerator '%N' cannot be declared in this class*

An enumerator cannot be declared with the same name as its containing class.

Example:

```
struct S {
    enum E {
    S,  // Error!
    T
    };
};
```

286 *static member '%N' cannot be declared in this class*

A static member cannot be declared with the same name as its containing class.

Example:

```
struct S {
    static int S;   // Error!
};
```

287 *constructor cannot have a return type*

A constructor cannot have a return type (even ***void***). The constructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:

```
class C {
public:
    C& C( int );
};
```

288 *constructor cannot be a static member*

A constructor is a special member function that takes raw storage and changes it into an instance of a class. In order to do this, it must have access to storage for the instance of the class so it is restricted to being a non-static member function.

Example:

```
class C {
public:
    static C( int );
};
```

289 *invalid copy constructor argument list (causes infinite recursion)*

A copy constructor's first argument must be a reference argument. Furthermore, any default arguments must also be reference arguments. Without the reference, a copy constructor would require a copy constructor to execute in order to prepare its arguments. Unfortunately, this would be calling itself since it is the copy constructor.

Example:

```
struct S {
    S( S const & );   // copy constructor
};
```

290 *constructor cannot be declared const or volatile*

A constructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

Example:

```
class C {
public:
    C( int ) const;
    C( float ) volatile;
};
```

291 *constructor cannot be virtual*

Virtual functions cannot be called for an object before it is constructed. For this reason, a virtual constructor is not allowed in the C++ language. Techniques for simulating a virtual constructor are known, one such technique is described in the ARM p.263.

Example:

```
class C {
public:
    virtual C( int );
};
```

292 *types do not match in simple type destructor*

A simple type destructor is available for "destructing" simple types. The destructor has no effect. Both of the types must be identical, for the destructor to have meaning.

Example:

```
void foo( int *p )
{
    p->int::~double();
}
```

293 *overloaded operator is ambiguous for operands used*

The Open Watcom C++ compiler performs exhaustive analysis using formalized techniques in order to decide what implicit conversions should be applied for overloading operators. Because of this, Open Watcom C++ detects ambiguities that may escape other C++ compilers. The most common ambiguity that Open Watcom C++ detects involves classes having constructors with single arguments and a user-defined conversion.

Example:

```
struct S {
    S(int);
    operator int();
    int a;
};

int fn( int b, int i, S s )
{
    //    i    : s.operator int()
    // OR S(i) : s
    return b ? i : s;
}
```

In the above example, "i" and "s" must be brought to a common type. Unfortunately, there are two common types so the compiler cannot decide which one it should choose, hence an ambiguity.

294 *feature not implemented*

The compiler does not support the indicated feature.

295 *invalid friend declaration*

This message indicates that the compiler found extra declaration specifiers like ***auto, float,*** or ***const*** in the friend declaration.

Example:

```
class C
{
    friend float;
};
```

296 *friend declarations may only be declared in a class*

This message indicates that a ***friend*** declaration was found outside a class scope (i.e., a class definition). Friends are only meaningful for class types.

Example:

```
extern void foo();
friend void foo();
```

297 *class friend declaration needs 'class' or 'struct' keyword*

The C++ language has evolved to require that all friend class declarations be of the form "class S" or "struct S". The Open Watcom C++ compiler accepts the older syntax with a warning but rejects the syntax in pure ISO/ANSI C++ mode.

Example:

```
struct S;
struct T {
    friend S;   // should be "friend class S;"
};
```

298 *class friend declarations cannot contain a class definition*

A class friend declaration cannot define a new class. This is a restriction required in the C++ language.

Example:

```
struct S {
    friend struct X {
    int f;
    };
};
```

299 *'%T' has already been declared as a friend*

The class in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
class S;
class T {
    friend class S;
    int tv;
    friend class S;
};
```

300 *function '%S' has already been declared as a friend*

The function in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
extern void foo();
class T {
    friend void foo();
    int tv;
    friend void foo();
};
```

301 *'friend', 'virtual' or 'inline' modifiers are not part of a function's type*

This message indicates that the modifiers may be incorrectly placed in the declaration. If the declaration is intended, it cannot be accepted because the modifiers can only be applied to functions that have code associated with them.

Example:

```
typedef friend (*PF)( void );
```

302 *cannot assign right expression to element on left*

This message indicates that the assignment cannot be performed. It usually arises in assignments of a class type to an arithmetic type.

Example:

```
struct S
{   int sv;
};
S s;
int foo()
{
    int k;
    k = s;
    return k;
}
```

303 *constructor is ambiguous for operands used*

The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S x = S(1.0);
```

304 *class '%s' has not been defined*

The name before a '::' scope resolution operator must be defined unless a member pointer is being declared.

Example:

```
struct S;

int S::* p; // OK
int S::a = 1;   // Error!
```

305 *all bit-fields in a union must be named*

This is a restriction in the C++ language. The same effect can be achieved with a named bitfield.

Example:

```
union u
{   unsigned bit1 :10;
    unsigned :6;
};
```

306 *cannot convert expression to type of cast*

The cast is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended cast.

Example:

```
struct T {
};

void fn()
{
    T y = (T) 0;
}
```

307 *conversion ambiguity: [expression] to [cast type]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

void fn()
{
    S x = (S) 1.0;
}
```

308 *an anonymous class without a declarator is useless*

There is no way to reference the type in this kind of declaration. A name must be provided for either the class or a variable using the class as its type.

Example:

```
struct {
    int a;
    int b;
};
```

309 *global anonymous union must be declared static*

This is a restriction in the C++ language. Since there is no unique name for the anonymous union, it is difficult for C++ translators to provide a correct implementation of external linkage anonymous unions.

Example:

```
static union {
    int a;
    int b;
};
```

310 *anonymous struct/union cannot have storage class in this context*

Anonymous unions (or structs) declared in class scopes cannot be ***static.*** Any other storage class is also disallowed.

Example:

```
struct S {
    static union {
    int iv;
    unsigned us;
    };
};
```

311 *union contains a protected member*

A union cannot have a ***protected*** member because a union cannot be a base class.

Example:

```
static union {
    int iv;
protected:
    unsigned sv;
} u;
```

312 *anonymous struct/union contains a private member '%S'*

An anonymous union (or struct) cannot have member functions or friends so it cannot have ***private*** members since no code could access them.

Example:

```
static union {
    int iv;
private:
    unsigned sv;
};
```

313 *anonymous struct/union contains a function member '%S'*

An anonymous union (or struct) cannot have any function members. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    void foo();     // error
    unsigned sv;
};
```

314 *anonymous struct/union contains a typedef member '%S'*

An anonymous union (or struct) cannot have any nested types. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    unsigned sv;
    typedef float F;
    F fv;
};
```

315 *anonymous struct/union contains an enumeration member '%S'*

An anonymous union (or struct) cannot have any enumeration members. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    enum choice { good, bad, indifferent };
    choice c;
    unsigned sv;
};
```

316 *anonymous struct/union member '%s' is not distinct in enclosing scope*

Since an anonymous union (or struct) provides its member names to the enclosing scope, the names must not collide with other names in the enclosing scope.

Example:

```
int iv;
unsigned sv;
static union {
    int iv;
    unsigned sv;
};
```

317 *unions cannot have members with destructors*

A union should only be used to organize memory in C++. Allowing union members to have destructors would mean that the same piece of memory could be destructed twice.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    ~T();
};
static union
{
    S su;
    T tu;
};
```

318 *unions cannot have members with user-defined assignment operators*

A union should only be used to organize memory in C++. Allowing union members to have assignment operators would mean that the same piece of memory could be assigned twice.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    int tv;
    operator = ( int );
    operator = ( float );
};
static union
{
    S su;
    T tu;
} u;
```

319 *anonymous struct/union cannot have any friends*

An anonymous union (or struct) cannot have any friends. This is a restriction in the C++ language.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
static union {
    S su1;
    S su2;
    friend class S;
};
```

320 *specific versions of template classes can only be defined in file scope*

Currently, specific versions of class templates can only be declared at file scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

Example:

```
template <class G> class S {
    G x;
};

struct Q {
    struct S<int> {
    int x;
    };
};

void foo()
{
    struct S<double> {
    double x;
    };
}
```

321 *anonymous union in a function may only be static or auto*

The current C++ language definition only allows ***auto*** anonymous unions. The Open Watcom C++ compiler allows ***static*** anonymous unions. Any other storage class is not allowed.

322 *static data members are not allowed in a local class*

Static data members are not allowed in a local class because there is no way to define the static member in file scope.

Example:

```
int foo()
{
    struct local {
    static int s;
    };

    local lv;

    lv.s = 3;
    return lv.s;
}
```

323 *conversion ambiguity: [return value] to [return type of function]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S fn()
{
    return 1.0;
}
```

324 *conversion of return value is impossible*

The return is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended return type.

Example:

```
struct T {
};

T fn()
{
    return 0;
}
```

325 *function cannot return a pointer based on __self*

A function cannot return a pointer that is based on ***__self.***

Example:

```
void __based(__self) *fn( unsigned );
```

326 *defining '%S' is not possible because its type has unknown size*

In order to define a variable, the size must be known so that the correct amount of storage can be reserved.

Example:

```
class S;
S sv;
```

327 *typedef cannot be initialized*

Initializing a ***typedef*** is meaningless in the C++ language.

Example:

```
typedef int INT = 15;
```

328 *storage class of '%S' conflicts with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to storage class. A symbol cannot be both ***static*** and ***extern.***

329 *modifiers of '%S' conflict with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to modifiers. Correct the program by using the same modifiers for both declarations.

330 *function cannot be initialized*

A function cannot be initialized with an initializer syntax intended for variables. A function body is the only way to provide a definition for a function.

331 *access permission of nested class '%T' conflicts with previous declaration*

Example:

```
struct S {
    struct N;   // public
private:
    struct N {  // private
    };
};
```

332 **** FATAL *** internal error in front end*

If this message appears, please report the problem directly to the Open Watcom development team. See http://www.openwatcom.org/.

333 *cannot convert argument to type specified in function prototype*

It is impossible to convert the indicated argument in the function.

Example:

```
extern int foo( int& );

extern int m;
extern int n;

int k = foo( m + n );
```

In the example, the value of "m+n" cannot be converted to a reference (it could be converted to a constant reference), as shown in the following example.

Example:

```
extern int foo( const int& );

extern int m;
extern int n;

int k = foo( m + n );
```

334 *conversion ambiguity: [argument] to [argument type in prototype]*

An argument in the function call could not be converted since there is more than one constructor or user-defined conversion which could be used to convert the argument.

Example:

```
struct S;

struct T
{
    T( S& );
};

struct S
{
    operator T();
};

S s;
extern int foo( T );
int k = foo( s );   // ambiguous
```

In the example, the argument "s" could be converted by both the constructor in class "T" and by the user-conversion in class "S".

335 *cannot be based on based pointer '%S'*

A based pointer cannot be based on another based pointer.

Example:

```
__segment s;
void __based(s) *p;
void __based(p) *q;
```

336 *declaration specifiers are required to declare '%N'*

The compiler has detected that the name does not represent a function. Only function declarations can leave out declaration specifiers. This error also shows up when a typedef name declaration is missing.

Example:

```
x;
typedef int;
```

337 *static function declared in block scope*

The C++ language does not allow static functions to be declared in block scope. This error can be triggered when the intent is to define a ***static*** variable. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as function prototypes. A work-around for this problem is contained in the example.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    static S a( C() );  // function prototype!
    static S b( (C()) );// variable definition
}
```

338 *cannot define a __based reference*

A C++ reference cannot be based on anything. Based modifiers can only be used with pointers.

Example:

```
__segment s;
void fn( int __based(s) & x );
```

339 *conversion ambiguity: conversion to common pointer type*

A conversion to a common base class of two different pointers has been attempted. The pointer conversion could not be performed because the destination type points to an ambiguous base class of one of the source types.

340 *cannot construct object from argument(s)*

There is not an appropriate constructor for the set of arguments provided.

341 *number of arguments for function '%S' is incorrect*

The number of arguments in the function call does not match the number declared for the indicated non-overloaded function.

Example:

```
extern int foo( int, int );
int k = foo( 1, 2, 3 );
```

In the example, the function was declared to have two arguments. Three arguments were used in the call.

342 *private base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

extern Der *pd;
Priv *pp = (Priv*)pd;
```

343 *private base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

Priv *foo( Der *p )
{
    return p;
}
```

344 *cannot subtract pointers to different objects*

Pointer subtraction can be performed only for objects of the same type.

Example:

```
#include <stddef.h>
ptrdiff_t diff( float *fp, int *ip )
{
    return fp - ip;
}
```

In the example, a diagnostic results from the attempt to subtract a pointer to an ***int*** object from a pointer to a ***float*** object.

345 *private base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

int foo( Der *pd, Priv *pp )
{
    return pd == pp;
}
```

346 *protected base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

extern Der *pd;
Prot *pp = (Prot*)pd;
```

347 *protected base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

Prot *foo( Der *p )
{
    return p;
}
```

348 *cannot define a member pointer with a memory model modifier*

A member pointer describes how to access a field from a class. Because of this a member pointer must be independent of any memory model considerations.

Example:

```
struct S;

int near S::*mp;
```

349 *protected base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

int foo( Der *pd, Prot *pp )
{
    return pd == pp;
}
```

350 *non-type parameter supplied for a type argument*

A non-type parameter (e.g., an address or a constant expression) has been supplied for a template type argument. A type should be used instead.

351 *type parameter supplied for a non-type argument*

A type parameter (e.g., ***int***) has been supplied for a template non-type argument. An address or a constant expression should be used instead.

352 *cannot access enclosing function's auto variable '%S'*

A local class member function cannot access its enclosing function's automatic variables.

Example:

```
void goop( void )
{
    int a;
    struct S
    {
    int foo( int c, int b )
        {
        return b + c + a;
        };
    };
}
```

353 *cannot initialize pointer to non-constant with a pointer to constant*

A pointer to a non-constant type cannot be initialized with a pointer to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:

```
extern const int *pic;
extern int *pi = pic;
```

354 *pointer expression is always >= 0*

The indicated pointer expression will always be true because the pointer value is always treated as an unsigned quantity, which will be greater or equal to zero.

Example:

```
extern char *p;
unsigned k = ( 0 <= p );     // always 1
```

355 *pointer expression is never < 0*

The indicated pointer expression will always be false because the pointer value is always treated as an unsigned quantity, which will be greater or equal zero.

Example:

```
extern char *p;
unsigned k = ( 0 >= p );    // always 0
```

356 *type cannot be used in this context*

This message is issued when a type name is being used in a context where a non-type name should be used.

Example:

```
struct S {
    typedef int T;
};

void fn( S *p )
{
    p->T = 1;
}
```

357 *virtual function may only be declared in a class*

Virtual functions can only be declared inside of a class. This error may be a result of forgetting the "C::" qualification of a virtual function's name.

Example:

```
virtual void foo();
struct S
{
    int f;
    virtual void bar();
};
virtual void bar()
{
    f = 9;
}
```

358 *'%T' referenced as a union*

A class type defined as a ***class*** or ***struct*** has been referenced as a ***union*** (i.e., union S).

Example:

```
struct S
{
    int s1, s2;
};
union S var;
```

359 *union '%T' referenced as a class*

A class type defined as a ***union*** has been referenced as a ***struct*** or a ***class*** (i.e., class S).

Example:

```
union S
{
    int s1, s2;
};
struct S var;
```

360 *typedef '%N' defined without an explicit type*

The typedef declaration was found to not have an explicit type in the declaration. If ***int*** is the desired type, use an explicit ***int*** keyword to specify the type.

Example:

```
typedef T;
```

361 *member function was not defined in its class*

Member functions of local classes must be defined in their class if they will be defined at all. This is a result of the C++ language not allowing nested function definitions.

Example:

```
void fn()
{
    struct S {
    int bar();
    };
}
```

362 *local class can only have its containing function as a friend*

A local class can only be referenced from within its containing function. It is impossible to define an external function that can reference the type of the local class.

Example:

```
extern void ext();
void foo()
{
    class S
    {
    int s;
    public:
    friend void ext();
    int q;
    };
}
```

363 *local class cannot have '%S' as a friend*

The only classes that a local class can have as a friend are classes within its own containing scope.

Example:

```
struct ext
{
    goop();
};
void foo()
{
    class S
    {
    int s;
    public:
    friend class ext;
    int q;
    };
}
```

364 *adjacent >=, <=, >, < operators*

This message is warning about the possibility that the code may not do what was intended. An expression like "a > b > c" evaluates one relational operator to a 1 or a 0 and then compares it against the other variable.

Example:

```
extern int a;
extern int b;
extern int c;
int k = a > b > c;
```

365 *cannot access enclosing function's argument '%S'*

A local class member function cannot access its enclosing function's arguments.

Example:

```
void goop( int d )
{
    struct S
    {
    int foo( int c, int b )
        {
        return b + c + d;
        };
    };
}
```

366 *support for switch '%s' is not implemented*

Actions for the indicated switch have not been implemented. The switch is supported for compatibility with the Open Watcom C compiler.

367 *conditional expression in if statement is always true*

The compiler has detected that the expression will always be true. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially true expressions.

Example:

```
#define TEST 143
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

368 *conditional expression in if statement is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

Example:

```
#define TEST 14-14
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

369 *selection expression in switch statement is a constant value*

The expression in the ***switch*** statement is a constant. This means that only one case label will be executed. If this is not the expected behaviour, check the switch expression.

Example:

```
#define TEST 0
int foo( int a, int b )
{
    switch ( TEST ) {
      case 0:
    return a;
      default:
    return b;
    }
}
```

370 *constructor is required for a class with a const member*

If a class has a constant member, a constructor is required in order to initialize it.

Example:

```
struct S
{
    const int s;
    int i;
};
```

371 *constructor is required for a class with a reference member*

If a class has a reference member, a constructor is required in order to initialize it.

Example:

```
struct S
{
    int& r;
    int i;
};
```

372 *inline member friend function '%S' is not allowed*

A friend that is a member function of another class cannot be defined. Inline friend rules are currently in flux so it is best to avoid inline friends.

373 *invalid modifier for auto variable*

An automatic variable cannot have a memory model adjustment because they are always located on the stack (or in a register). There are also other types of modifiers that are not allowed for auto variables such as thread-specific data modifiers.

Example:

```
int fn( int far x )
{
    int far y = x + 1;
    return y;
}
```

374 *object (or object pointer) required to access non-static data member*

A reference to a member in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m;
    static void fn()
    {
    m = 1;  // Error!
    }
};
```

375 *user-defined conversion has not been declared*

The named user-defined conversion has not been declared in the class of any of its base classes.

Example:

```
struct S {
    operator int();
    int a;
};

double fn( S *p )
{
    return p->operator double();
}
```

376 *virtual function must be a non-static member function*

A member function cannot be both a ***static*** function and a ***virtual*** function. A static member function does not have a ***this*** argument whereas a ***virtual*** function must have a ***this*** argument so that the virtual function table can be accessed in order to call it.

Example:

```
struct S
{
    static virtual int foo();   // error
    virtual int bar();      // ok
    static int stat();      // ok
};
```

377 *protected base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
protected:
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is protected.

378 *private base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is private.

379 *delete expression will invoke a non-virtual destructor*

In C++, it is possible to assign a base class pointer the value of a derived class pointer so that code that makes use of base class virtual functions can be used. A problem that occurs is that a ***delete*** has to know the correct size of the type in some instances (i.e., when a two argument version of ***operator delete*** is defined for a class). This problem is solved by requiring that a destructor be defined as ***virtual*** if polymorphic deletes must work. The ***delete*** expression will virtually call the correct destructor, which knows the correct size of the complete object. This message informs you that the class you are deleting has virtual functions but it has a non-virtual destructor. This means that the delete will not work correctly in all circumstances.

Example:

```
#include <stddef.h>

struct B {
    int b;
    void operator delete( void *, size_t );
    virtual void fn();
    ~B();
};
struct D : B {
    int d;
    void operator delete( void *, size_t );
    virtual void fn();
    ~D();
};

void dfn( B *p )
{
    delete p;   // could be a pointer to D!
}
```

380 *'offsetof' is not allowed for a function*

A member function does not have an offset like simple data members. If this is required, use a member pointer.

Example:

```
#include <stddef.h>

struct S
{
    int fun();
};

int s = offsetof( S, fun );
```

381 *'offsetof' is not allowed for an enumeration*

An enumeration does not have an offset like simple data members.

Example:

```
#include <stddef.h>

struct S
{
    enum SE { S1, S2, S3, S4 };
    SE var;
};

int s = offsetof( S, SE );
```

382 *could not initialize for code generation*

The source code has been parsed and fully analysed when this error is emitted. The compiler attempted to start generating object code but due to some problem (e.g., out of memory, no file handles) could not initialize itself. Try changing the compilation environment to eliminate this error.

383 *'offsetof' is not allowed for an undefined type*

The class type used in ***offsetof*** must be completely defined, otherwise data member offsets will not be known.

Example:

```
#include <stddef.h>

struct S {
    int a;
    int b;
    int c[ offsetof( S, b ) ];
};
```

384 *attempt to override virtual function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading. The exception to this rule in the C++ language involves restricted changes in the return type of virtual functions. The derived virtual function's return type can be derived from the return type of the base virtual function.

Example:

```
struct B {
    virtual B *fn();
};
struct D : B {
    virtual D *fn();
};
```

385 *attempt to overload function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading.

Example:

```
int foo( char );
unsigned foo( char );
```

386 *attempt to use pointer to undefined class*

An attempt was made to indirect or increment a pointer to an undefined class. Since the class is undefined, the size is not known so the compiler cannot compile the expression properly.

Example:

```
class C;
extern C* pc1;
C* pc2 = ++pc1;      // C not defined

int foo( C*p )
{
    return p->x;     // C not defined
}
```

387 *expression is useful only for its side effects*

The indicated expression is not meaningful. The expression, however, does contain one or more side effects.

Example:

```
extern int* i;
void func()
{
    *(i++);
}
```

In the example, the expression is a reference to an integer which is meaningless in itself. The incrementation of the pointer in the expression is a side effect.

388 *integral constant will be truncated during assignment or initialization*

This message indicates that the compiler knows that a constant value will not be preserved after the assignment. If this is acceptable, cast the constant value to the appropriate type in the assignment.

Example:

```
unsigned char c = 567;
```

389 *integral value may be truncated during assignment or initialization*

This message indicates that the compiler knows that all values will not be preserved after the assignment. If this is acceptable, cast the value to the appropriate type in the assignment.

Example:

```
extern unsigned s;
unsigned char c = s;
```

390 *cannot generate default constructor to initialize '%T' since constructors were declared*

A default constructor will not be generated by the compiler if there are already constructors declared. Try using default arguments to change one of the constructors to a default constructor or define a default constructor explicitly.

Example:

```
class C {
    C( const C& );
public :
    int c;
};
C cv;
```

391 *assignment found in boolean expression*

This is a construct that can lead to errors if it was intended to be an equality (using "==") test.

Example:

```
int foo( int a, int b )
{
    if( a = b ) {
    return b;
    }
    return a;       // always return 1 ?
}
```

392 *definition: '%F'*

This informational message indicates where the symbol in question was defined. The message is displayed following an error or warning diagnostic for the symbol in question.

Example:

```
static int a = 9;
int b = 89;
```

The variable 'a' is not referenced in the preceding example and so will cause a warning to be generated. Following the warning, the informational message indicates the line at which 'a' was declared.

393 *included from %s(%u)*

This informational message indicates the line number of the file including the file in which an error or warning was diagnosed. A number of such messages will allow you to trace back through the ***#include*** directives which are currently being processed.

394 *reference object must be initialized*

A reference cannot be set except through initialization. Also references cannot be 0 so they must always be initialized.

Example:

```
int & ref;
```

395 *option requires an identifier*

The specified option is not recognized by the compiler since there was no identifier after it (i.e., "-nt=module").

396 *'main' cannot be overloaded*

There can only be one entry point for a C++ program. The "main" function cannot be overloaded.

Example:

```
int main();
int main( int );
```

397 *'new' expression cannot allocate a void*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot allocate an instance of ***void.***

Example:

```
void *p = new void;
```

398 *'new' expression cannot allocate a function*

A function type cannot be allocated since there is no meaningful size that can be used. The ***new*** expression can allocate a pointer to a function.

Example:

```
typedef int tdfun( int );
tdfun *tdv = new tdfun;
```

399 *'new' expression allocates a const or volatile object*

The pool of raw memory cannot be guaranteed to support ***const*** or ***volatile*** semantics. Usually ***const*** and ***volatile*** are used for statically allocated objects.

Example:

```
typedef const int con_int;
con_int* p = new con_int;
```

400 *cannot convert right expression for initialization*

The initialization is trying to convert an argument expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended conversion.

Example:

```
struct T {
};

T x = 0;
```

401 *conversion ambiguity: [initialization expression] to [type of object]*

The initialization caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S x = 1.0;
```

402 *class template '%S' has already been declared as a friend*

The class template in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
template <class T>
    class S;

class X {
    friend class S;
    int f;
    friend class S;
};
```

403 *private base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

404 *protected base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

405 *cannot return a pointer or reference to a constant object*

A pointer or reference to a constant object cannot be returned.

Example:

```
int *foo( const int *p )
{
    return p;
}
```

406 *cannot pass a pointer or reference to a constant object*

A pointer or reference to a constant object could not be passed as an argument.

Example:

```
int *bar( int * );
int *foo( const int *p )
{
    return bar( p );
}
```

407 *class templates must be named*

There is no syntax in the C++ language to reference an unnamed class template.

Example:

```
template <class T>
    class {
    };
```

408 *function templates can only name functions*

Variables cannot be overloaded in C++ so it is not possible to have many different instances of a variable with different types.

Example:

```
template <class T>
    T x[1];
```

409 *template argument '%S' is not used in the function argument list*

This restriction ensures that function templates can be bound to types during overload resolution. Functions currently can only be overloaded based on argument types.

Example:

```
template <class T>
    int foo( int * );
template <class T>
    T bar( int * );
```

410 *destructor cannot be declared const or volatile*

A destructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

411 *static member function cannot be declared const or volatile*

A static member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

412 *only member functions can be declared const or volatile*

A non-member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

413 *'const' or 'volatile' modifiers are not part of a function's type*

The ***const*** and ***volatile*** qualifiers for a function cannot be used in typedefs or pointers to functions. The trailing qualifiers are used to change the type of the implicit ***this*** argument so that member functions that do not modify the object can be declared accurately.

Example:

```
// const is illegal
typedef void (*baddcl)() const;

struct S {
    void fun() const;
    int a;
};

// "this" has type "S const *"
void S::fun() const
{
    this->a = 1;    // Error!
}
```

414 *type cannot be defined in an argument*

A new type cannot be defined in an argument because the type will only be visible within the function. This amounts to defining a function that can never be called because C++ uses name equivalence for type checking.

Example:

```
extern foo( struct S { int s; } );
```

415 *type cannot be defined in return type*

This is a restriction in the current C++ language. A function prototype should only use previously declared types in order to guarantee that it can be called from other functions. The restriction is required for templates because the compiler would have to wait until the end of a class definition before it could decide whether a class template or function template is being defined.

Example:

```
template <class T>
    class C {
    T value;
    } fn( T x ) {
    C y;

    y.x = 0;
    return y;
    };
```

A common problem that results in this error is to forget to terminate a class or enum definition with a semicolon.

Example:

```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

416 *data members cannot be initialized inside a class definition*

This message appears when an initialization is attempted inside of a class definition. In the case of static data members, initialization must be done outside the class definition. Ordinary data members can be initialized in a constructor.

Example:

```
struct S {
    static const int size = 1;
};
```

417 *only virtual functions may be declared pure*

The C++ language requires that all pure functions be declared virtual. A pure function establishes an interface that must consist of virtual functions because the functions are required to be defined in the derived class.

Example:

```
struct S {
    void foo() = 0;
};
```

418 *destructor is not declared in its proper class*

The destructor name is not declared in its own class or qualified by its own class. This is required in the C++ language.

419 *cannot call non-const function for a constant object*

A function that does not promise to not modify an object cannot be called for a constant object. A function can declare its intention to not modify an object by using the ***const*** qualifier.

Example:

```
struct S {
    void fn();
};

void cfn( const S *p )
{
    p->fn();    // Error!
}
```

420 *memory initializer list may only appear in a constructor definition*

A memory initializer list should be declared along with the body of the constructor function.

421 *cannot initialize member '%N' twice*

A member cannot be initialized twice in a member initialization list.

422 *cannot initialize base class '%T' twice*

A base class cannot be constructed twice in a member initialization list.

423 *'%T' is not a direct base class*

A base class initializer in a member initialization list must either be a direct base class or a virtual base class.

424 *'%N' cannot be initialized because it is not a member*

The name used in the member initialization list does not name a member in the class.

425 *'%N' cannot be initialized because it is a member function*

The name used in the member initialization list does not name a non-static data member in the class.

426 *'%N' cannot be initialized because it is a static member*

The name used in the member initialization list does not name a non-static data member in the class.

427 *'%N' has not been declared as a member*

This message indicates that the member does not exist in the qualified class. This usually occurs in the context of access declarations.

428 *const/reference member '%S' must have an initializer*

The ***const*** or reference member does not have an initializer so the constructor is not completely defined. The member initialization list is the only way to initialize these types of members.

429 *abstract class '%T' cannot be used as an argument type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as an argument type.

430 *abstract class '%T' cannot be used as a function return type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as a return type.

431 *defining '%S' is not possible because '%T' is an abstract class*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as either a member or a variable.

432 *cannot convert to an abstract class '%T'*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as the destination type in a conversion.

433 *mangled name for '%S' has been truncated*

The name used in the object file that encodes the name and full type of the symbol is often called a mangled name. The warning indicates that the mangled name had to be truncated due to limitations in the object file format.

434 *cannot convert to a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

435 *cannot convert a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

436 *cannot construct an abstract class*

An instance of an abstract class cannot be created because an abstract class can only be used as a base class.

437 *cannot construct an undefined class*

An instance of an undefined class cannot be created because the size is not known.

438 *string literal concatenated during array initialization*

This message indicates that a missing comma (’,’) could have made a quiet change in the program. Otherwise, ignore this message.

439 *maximum size of segment ’%s’ has been exceeded for ’%S’*

The indicated symbol has grown in size to a point where it has caused the segment it is defined inside of to be exhausted.

440 *maximum data item size has been exceeded for ’%S’*

A non-huge data item is larger than 64k bytes in size. This message only occurs during 16-bit compilation of C++ code.

441 *function attribute has been repeated*

A function attribute (like the ***__export*** attribute) has been repeated. Remove the extra attribute to correct the declaration.

442 *modifier has been repeated*

A modifier (like the ***far*** modifier) has been repeated. Remove the extra modifier to correct the declaration.

443 *illegal combination of memory model modifiers*

Memory model modifiers must be used individually because they cannot be combined meaningfully.

444 *argument name ’%N’ has already been used*

The indicated argument name has already been used in the same argument list. This is not allowed in the C++ language.

445 *function definition for ’%S’ must be declared with an explicit argument list*

A function cannot be defined with a typedef. The argument list must be explicit.

446 *user-defined conversion cannot convert to its own class or base class*

A user-defined conversion cannot be declared as a conversion either to its own class or to a base class of itself.

Example:

```
struct B {
};
struct D : private B {
    operator B();
};
```

447 *user-defined conversion cannot convert to void*

A user-defined conversion cannot be declared as a conversion to ***void.***

Example:

```
struct S {
    operator void();
};
```

448 *expecting identifier*

An identifier was expected during processing.

449 *symbol '%S' does not have a segment associated with it*

A pointer cannot be based on a member because it has no segment associated with it. A member describes a layout of storage that can occur in any segment.

450 *symbol '%S' must have integral or pointer type*

If a symbol is based on another symbol, it must be integral or a pointer type. An integral type indicates the segment value that will be used. A pointer type means that all accesses will be added to the pointer value to construct a full pointer.

451 *symbol '%S' cannot be accessed in all contexts*

The symbol that the pointer is based on is in another class so it cannot be accessed in all contexts that the based pointer can be accessed.

452 *cannot convert class expression to be copied*

A convert class expression could not be copied.

453 *conversion ambiguity: multiple copy constructors*

More than one constructor could be used to copy a class object.

454 *function template '%S' already has a definition*

The function template has already been defined with a function body. A function template cannot be defined twice even if the function body is identical.

Example:

```
template <class T>
    void f( T *p )
    {
    }
template <class T>
    void f( T *p )
    {
    }
```

455 *function templates cannot have default arguments*

A function template must not have default arguments because there are certain types of default arguments that do not force the function argument to be a specific type.

Example:

```
template <class T>
    void f2( T *p = 0 )
    {
    }
```

456 *'main' cannot be a function template*

This is a restriction in the C++ language because "main" cannot be overloaded. A function template provides the possibility of having more than one "main" function.

457 *'%S' was previously declared as a typedef*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:

```
typedef int T;
int T( int )        // error!
{
}

enum E { A, B, C };
void E()
{
    enum E x = A;   // use "enum E"
}

class C { };
void C()
{
    class C x;      // use "class C"
}
```

458 *'%S' was previously declared as a variable/function*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:

```
int T( int )
{
}
typedef int T;      // error!

void E()
{
}
enum E { A, B, C };

enum E x = A;       // use "enum E"

void C()
{
}
class C { };

class C x;      // use "class C"
```

459 *private base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

460 *protected base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

461 *maximum size of DGROUP has been exceeded for '%S' in segment '%s'*

The indicated symbol's size has caused the DGROUP contribution of this module to exceed 64k. Changing memory models or declaring some data as ***far*** data are two ways of fixing this problem.

462 *type of return value is not the enumeration type of function*

The return value does not have the proper enumeration type. Keep in mind that integral values are not automatically converted to enum types like the C language.

463 *linkage must be first in a declaration; probable cause: missing ';'*

This message usually indicates a missing semicolon (';'). The linkage specification must be the first part of a declaration if it is used.

464 *'main' cannot be a static function*

This is a restriction in the C++ language because "main" must have external linkage.

465 *'main' cannot be an inline function*

This is a restriction in the C++ language because "main" must have external linkage.

466 *'main' cannot be referenced*

This is a restriction in the C++ language to prevent implementations from having to work around multiple invocations of "main". This can occur if an implementation has to generate special code in "main" to construct all of the statically allocated classes.

467 *cannot call a non-volatile function for a volatile object*

A function that does not promise to not modify an object using ***volatile*** semantics cannot be called for a volatile object. A function can declare its intention to modify an object only through ***volatile*** semantics by using the ***volatile*** qualifier.

Example:

```
struct S {
    void fn();
};

void cfn( volatile S *p )
{
    p->fn();    // Error!
}
```

468 *cannot convert pointer to constant or volatile objects to pointer to void*

You cannot convert a pointer to constant or volatile objects to 'void*'.

Example:

```
extern const int* pci;
extern void *vp;

int k = ( pci == vp );
```

469 *cannot convert pointer to constant or non-volatile objects to pointer to volatile void*

You cannot convert a pointer to constant or non-volatile objects to 'volatile void*'.

Example:

```
extern const int* pci;
extern volatile void *vp;

int k = ( pci == vp );
```

470 *address of function is too large to be converted to pointer to void*

The address of a function can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the function pointer.

Example:

```
void __far foo();
void __near *v = &foo;
```

471 *address of data object is too large to be converted to pointer to void*

The address of an object can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the pointer.

Example:

```
int __far *ip;
void __near *v = ip;
```

472 *expression with side effect in sizeof discarded*

The indicated expression will be discarded; consequently, any side effects in that expression will not be executed.

Example:

```
int a = 14;
int b = sizeof( a++ );
```

In the example, the variable `a` will still have a value 14 after `b` has been initialized.

473 *function argument(s) do not match those in prototype*

The C++ language requires great precision in specifying arguments for a function. For instance, a pointer to `char` is considered different than a pointer to `unsigned char` regardless of whether `char` is an unsigned quantity. This message occurs when a non-overloaded function is invoked and one or more of the arguments cannot be converted. It also occurs when the number of arguments differs from the number specified in the prototype.

474 *conversion ambiguity: [expression] to [class object]*

The conversion of the expression to a class object is ambiguous.

475 *cannot assign right expression to class object*

The expression on the right cannot be assigned to the indicated class object.

476 *argument count is %d since there is an implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function with a ***this*** argument so it may have one more argument than expected.

477 *argument count is %d since there is no implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function without a ***this*** argument so it may have one less argument than expected.

478 *argument count is %d for a non-member function*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is not a member function but it could be declared as a ***friend*** function.

479 *conversion ambiguity: multiple copy constructors to copy array '%S'*

More than one constructor to copy the indicated array exists.

480 *variable/function has the same name as the class/enum '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the typedef name was declared in another file. This usually means that there are two unrelated uses of the same name.

481 *class/enum has the same name as the function/variable '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the function/variable name was declared in another file. This usually means that there are two unrelated uses of the same name. Furthermore, all references to the class or enum must be elaborated (i.e., use 'class C' instead of 'C') in order for subsequent references to compile properly.

482 *cannot create a default constructor*

A default constructor could not be created, because other constructors were declared for the class in question.

Example:

```
struct X {
    X(X&);
};
struct Y {
    X a[10];
};
Y yvar;
```

In the example, the variable "yvar" causes a default constructor for the class "Y" to be generated. The default constructor for "Y" attempts to call the default constructor for "X" in order to initialize the array "a" in class "Y". The default constructor for "X" cannot be defined because another constructor has been declared.

483 *attempting to access default constructor for %T*

This informational message indicates that a default constructor was referenced but could not be generated.

484 *cannot align symbol '%S' to segment boundary*

The indicated symbol requires more than one segment of storage and the symbol's components cannot be aligned to the segment boundary.

485 *friend declaration does not specify a class or function*

A class or function must be declared as a friend.

Example:

```
struct T {
    // should be class or function declaration
    friend int;
};
```

486 *cannot take address of overloaded function*

This message indicates that an overloaded function's name was used in a context where a final type could not be found. Because a final type was not specified, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the expression.

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );
int k = ( p == &foo );          // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );

// introduce temporary
static int (*temp)( char ) = &foo;

// ok
int k = ( p == temp );
```

487 *cannot use address of overloaded function as a variable argument*

This message indicates that an overloaded function's name was used as a argument for a "..." style function. Because a final function type is not present, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the call.

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );
int k = ellip_fun( 14, &foo );      // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );

static int (*temp)( char ) = &foo;  // introduce
temporary

int k = ellip_fun( 14, temp );      // ok
```

488 *'%N' cannot be overloaded*

The indicated function cannot be overloaded. Functions that fall into this category include ***operator delete.***

489 *symbol '%S' has already been initialized*

The indicated symbol has already been initialized. It cannot be initialized twice even if the initialization value is identical.

490 *delete expression is a pointer to a function*

A pointer to a function cannot be allocated so it cannot be deleted.

491 *delete of a pointer to const data*

Since deleting a pointer may involve modification of data, it is not always safe to delete a pointer to const data.

Example:

```
struct S { };
void fn( S const *p, S const *q ) {
    delete p;
    delete [] q;
}
```

492 *delete expression is not a pointer to data*

A ***delete*** expression can only delete pointers. For example, trying to delete an ***int*** is not allowed in the C++ language.

Example:

```
void fn( int a )
{
    delete a;   // Error!
}
```

493 *template argument is not a constant expression*

The compiler has found an incorrect expression provided as the value for a constant value template argument. The only expressions allowed for scalar template arguments are integral constant expressions.

494 *template argument is not an external linkage symbol*

The compiler has found an incorrect expression provided as the value for a pointer value template argument. The only expressions allowed for pointer template arguments are addresses of symbols. Any symbols must have external linkage or must be static class members.

495 *conversion of const reference to volatile reference*

The constant value can be modified by assigning into the volatile reference. This would allow constant data to be modified quietly.

Example:

```
void fn( const int &rci )
{
    int volatile &r = rci;  // Error!
}
```

496 *conversion of volatile reference to const reference*

The volatile value can be read incorrectly by accessing the const reference. This would allow volatile data to be accessed without correct volatile semantics.

Example:

```
void fn( volatile int &rvi )
{
    int const &r = rvi; // Error!
}
```

497 *conversion of const or volatile reference to plain reference*

The constant value can be modified by assigning into the plain reference. This would allow constant data to be modified quietly. In the case of volatile data, any access to the plain reference will not respect the volatility of the data and thus would be incorrectly accessing the data.

Example:

```
void fn( const int &rci, volatile int &rvi )
{
    int &r1 = rci;  // Error!
    int &r2 = rvi;  // Error!
}
```

498 *syntax error before '%s'; probable cause: incorrectly spelled type name*

The identifier in the error message has not been declared as a type name in any scope at this point in the code. This may be the cause of the syntax error.

499 *object (or object pointer) required to access non-static member function*

A reference to a member function in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m();
    static void fn()
    {
    m();    // Error!
    }
};
```

500 *object (or object pointer) cannot be used to access function*

The indicated object (or object pointer) cannot be used to access function.

501 *object (or object pointer) cannot be used to access data*

The indicated object (or object pointer) cannot be used to access data.

502 *cannot access member function in enclosing class*

A member function in enclosing class cannot be accessed.

503 *cannot access data member in enclosing class*

A data member in enclosing class cannot be accessed.

504 *syntax error before type name '%s'*

The identifier in the error message has been declared as a type name at this point in the code. This may be the cause of the syntax error.

505 *implementation restriction: cannot generate thunk from '%S'*

This implementation restriction is due to the use of a shared code generator between Open Watcom compilers. The virtual ***this*** adjustment thunks are generated as functions linked into the virtual function table. The functions rely on knowing the correct number of arguments to pass on to the overriding virtual function but in the case of ellipsis (...) functions, the number of arguments cannot be known when the thunk function is being generated by the compiler. The target symbol is listed in a diagnostic message. The work around for this problem is to recode the source so that the virtual functions make use of the va_list type found in the stdarg header file.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    virtual void fun( char *, ... );
};
struct D : B {
    virtual void fun( char *, ... );
};
void B::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
    cout << va_arg( args, char ) << endl;
    ++f;
    }
    va_end( args );
}
void D::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
    cout << va_arg( args, int ) << endl;
    ++f;
    }
    va_end( args );
}
```

The previous example can be changed to the following code with corresponding changes to the contents of the virtual functions.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    void fun( char *f, ... )
    {
    va_list args;

    va_start( args, f );
    _fun( f, args );
    va_end( args );
    }
    virtual void _fun( char *, va_list );
};
```

```
~b
struct D : B {
    // this can be removed since using B::fun
    // will result in the same behaviour
    // since _fun is a virtual function
    void fun( char *f, ... )
    {
    va_list args;

    va_start( args, f );
    _fun( f, args );
    va_end( args );
    }
    virtual void _fun( char *, va_list );
};
~b
void B::_fun( char *f, va_list args )
{
    while( *f ) {
    cout << va_arg( args, char ) << endl;
    ++f;
    }
}
~b
void D::_fun( char *f, va_list args )
{
    while( *f ) {
    cout << va_arg( args, int ) << endl;
    ++f;
    }
}

~b
// no changes are required for users of the class
B x;
D y;

void dump( B *p )
{
    p->fun( "1234", 'a', 'b', 'c', 'd' );
    p->fun( "12", 'a', 'b' );
}

~b
void main()
{
    dump( &x );
    dump( &y );
}
```

506 *conversion of __based(void) pointer to virtual base class*

An __based(void) pointer to a class object cannot be converted to a pointer to virtual base class, since this conversion applies only to specific objects.

Example:

```
struct Base {};
struct Derived : virtual Base {};
Derived __based( void ) *p_derived;
Base __based( void ) *p_base = p_derived; // error
```

The conversion would be allowed if the base class were not virtual.

507 *class for target operand is not derived from class for source operand*

A member pointer conversion can only be performed safely when converting a base class member pointer to a derived class member pointer.

508 *conversion ambiguity: [pointer to class member] to [assignment object]*

The base class in the original member pointer is not a unique base class of the derived class.

509 *conversion of pointer to class member involves a private base class*

The member pointer conversion required access to a private base class. The access check did not succeed so the conversion is not allowed.

510 *conversion of pointer to class member involves a protected base class*

The member pointer conversion required access to a protected base class. The access check did not succeed so the conversion is not allowed.

511 *item is neither a non-static member function nor data member*

A member pointer can only be created for non-static member functions and non-static data members. Static members can have their address taken just like their file scope counterparts.

512 *function address cannot be converted to pointer to class member*

The indicated function address cannot be converted to pointer to class member.

513 *conversion ambiguity: [address of function] to [pointer to class member]*

The indicated conversion is ambiguous.

514 *addressed function is in a private base class*

The addressed function is in a private base class.

515 *addressed function is in a protected base class*

The addressed function is in a protected base class.

516 *class for object is not defined*

The left hand operand for the "." or ".*" operator must be of a class type that is completely defined.

Example:

```
class C;

int fun( C& x )
{
    return x.y;     // class C not defined
}
```

517 *left expression is not a class object*

The left hand operand for the ".*" operator must be of a class type since member pointers can only be used with classes.

518 *right expression is not a pointer to class member*

The right hand operand for the ".*" operator must be a member pointer type.

519 *cannot convert pointer to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

520 *conversion ambiguity: [pointer] to [class of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

521 *conversion of pointer to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

522 *conversion of pointer to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

523 *cannot convert object to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

524 *conversion ambiguity: [object] to [class object of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

525 *conversion of object to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

526 *conversion of object to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

527 *conversion of pointer to class member from a derived to a base class*

A member pointer can only be converted from a base class to a derived class. This is the opposite of the conversion rule for pointers.

528 *form is '#pragma inline_recursion en' where 'en' is 'on' or 'off'*

This ***pragma*** indicates whether inline expansion will occur for an inline function which is called (possibly indirectly) a subsequent time during an inline expansion. Either 'on' or 'off' must be specified.

529 *expression for number of array elements must be integral*

The expression for the number of elements in a ***new*** expression must be integral because it is used to calculate the size of the allocation (which is an integral quantity). The compiler will not automatically convert to an integer because of rounding and truncation issues with floating-point values.

530 *function accessed with '.*' or '->*' can only be called*

The result of the ".*" and "->*" operators can only be called because it is often specific to the instance used for the left hand operand.

531 *left operand must be a pointer, pointer to class member, or arithmetic*

The left operand must be a pointer, pointer to class member, or arithmetic.

532 *right operand must be a pointer, pointer to class member, or arithmetic*

The right operand must be a pointer, pointer to class member, or arithmetic.

533 *neither pointer to class member can be converted to the other*

The two member pointers being compared are from two unrelated classes. They cannot be compared since their members can never be related.

534 *left operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the left operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( p == mp )
    p[0] = 1;
}
```

535 *right operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the right operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( mp == p )
    p[0] = 1;
}
```

536 *cannot use '.*' nor '->*' with pointer to class member with zero value*

The compiler has detected a NULL pointer use with a member pointer dereference.

537 *operand is not a valid pointer to class member*

The operand cannot be converted to a valid pointer to class member.

Example:

```
struct S;
int S::* fn()
{
    int a;
    return a;
}
```

538 *destructor can be invoked only with '.' or '->'*

This is a restriction in the C++ language. An explicit invocation of a destructor is not recommended for objects that have their destructor called automatically.

539 *class of destructor must be class of object being destructed*

Destructors can only be called for the exact static type of the object being destroyed.

540 *destructor is not properly qualified*

An explicit destructor invocation can only be qualified with its own class.

541 *pointers to class members reference different object types*

Conversion of member pointers can only occur if the object types are identical. This is necessary to ensure type safety.

542 *operand must be pointer to class or struct*

The left hand operand of a '->*' operator must be a pointer to a class. This is a restriction in the C++ language.

543 *expression must have void type*

If one operand of the ':' operator has ***void*** type, then the other operand must also have ***void*** type.

544 *expression types do not match for ':' operator*

The compiler could not bring both operands to a common type. This is necessary because the result of the conditional operator must be a unique type.

545 *cannot create an undefined type with 'operator new'*

A ***new*** expression cannot allocate an undefined type because it must know how large an allocation is required and it must also know whether there are any constructors to execute.

546 *delete of a pointer to an undefined type*

A ***delete*** expression cannot safely deallocate an undefined type because it must know whether there are any destructors to execute. In spite of this, the ISO/ANSI C++ Working Paper requires that an implementation support this usage.

Example:

```
struct U;

void foo( U *p, U *q ) {
    delete p;
    delete [] q;
}
```

547 *cannot access '%S' through a private base class*

The indicated symbol cannot be accessed because it requires access to a private base class.

548 *cannot access '%S' through a protected base class*

The indicated symbol cannot be accessed because it requires access to a protected base class.

549 *'sizeof' operand contains compiler generated information*

The type used in the 'sizeof' operand contains compiler generated information. Clearing a struct with a call to memset() would invalidate all of this information.

550 *cannot convert ':' operands to a common reference type*

The two reference types cannot be converted to a common reference type. This can happen when the types are not related through base class inheritance.

551 *conversion ambiguity: [reference to object] to [type of opposite ':' operand]*

One of the reference types is an ambiguous base class of the other. This prevents the compiler from converting the operand to a unique common type.

552 *conversion of reference to ':' object involves a private base class*

The conversion of the reference operands requires a conversion through a private base class.

553 *conversion of reference to ':' object involves a protected base class*

The conversion of the reference operands requires a conversion through a protected base class.

554 *expression must have type arithmetic, pointer, or pointer to class member*

This message means that the type cannot be converted to any of these types, also. All of the mentioned types can be compared against zero ('0') to produce a true or false value.

555 *expression for 'while' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

556 *testing expression for 'for' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

557 *message number '%d' is invalid*

The message number used in the #pragma does not match the message number for any warning message. This message can also indicate that a number or '*' (meaning all warnings) was not found when it was expected.

558 *warning level must be an integer in range 0 to 9*

The new warning level that can be used for the warning can be in the range 0 to 9. The level 0 means that the warning will be treated as an error (compilation will not succeed). Levels 1 up to 9 are used to classify warnings. The -w option sets an upper limit on the level for warnings. By setting the level above the command line limit, you effectively ignore all cases where the warning shows up.

559 *function '%S' cannot be defined because it is generated by the compiler*

The indicated function cannot be defined because it is generated by the compiler. The compiler will automatically generate default constructors, copy constructors, assignment operators, and destructors according to the rules of the C++ language. This message indicates that you did not declare the function in the class definition.

560 *neither environment variable nor file found for '@' name*

The indirection operator for the command line will first check for an environment variable of the name and use the contents for the command line. If an environment variable is not found, a check for a file with the same name will occur.

561 *more than 5 indirections during command line processing*

The Open Watcom C++ compiler only allows a fixed number nested indirections using files or environment variables, to prevent runaway chains of indirections.

562 *cannot take address of non-static member function*

The only way to create a value that described the non-static member function is to use a member pointer.

563 *cannot generate default '%S' because class contains either a constant or a reference member*

An assignment operator cannot be generated because the class contains members that cannot be assigned into.

564 *cannot convert pointer to non-constant or volatile objects to pointer to const void*

A pointer to non-constant or volatile objects cannot be converted to 'const void*'.

565 *cannot convert pointer to non-constant or non-volatile objects to pointer to const volatile void*

A pointer to non-constant or non-volatile objects cannot be converted to 'const volatile void*'.

566 *cannot initialize pointer to non-volatile with a pointer to volatile*

A pointer to a non-volatile type cannot be initialized with a pointer to a volatile type because this would allow volatile data to be modified without volatile semantics via the non-volatile pointer to it.

567 *cannot pass a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be passed in this context.

568 *cannot return a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be returned.

569 *left expression is not a pointer to a volatile object*

One cannot assign a pointer to a volatile type to a pointer to a non-volatile type. This would allow a volatile object to be modified via the non-volatile pointer. Use a cast if this is absolutely necessary.

570 *virtual function override for '%S' is ambiguous*

This message indicates that there are at least two overrides for the function in the base class. The compiler cannot arbitrarily choose one so it is up to the programmer to make sure there is an unambiguous choice. Two of the overriding functions are listed as informational messages.

571 *initialization priority must be number 0-255, 'library', or 'program'*

An incorrect module initialization priority has been provided. Check the User's Guide for the correct format of the priority directive.

572 *previous case label defined %L*

This informational message indicates where a preceding ***case*** label is defined.

573 *previous default label defined %L*

This informational message indicates where a preceding ***default*** label is defined.

574 *label defined %L*

This informational message indicates where a label is defined.

575 *label referenced %L*

This informational message indicates where a label is referenced.

576 *object thrown has type: %T*

This informational message indicates the type of the object being thrown.

577 *object thrown has an ambiguous base class %T*

It is illegal to throw an object with a base class to which a conversion would be ambiguous.

Example:

```
struct ambiguous{ };
struct base1 : public ambiguous { };
struct base2 : public ambiguous { };
struct derived : public base1, public base2 { };

foo( derived &object )
{
    throw object;
}
```

The ***throw*** will cause an error to be displayed because an object of type "derived" cannot be converted to an object of type "ambiguous".

578 *form is '#pragma inline_depth level' where 'level' is 0 to 255*

This ***pragma*** sets the number of times inline expansion will occur for an inline function which contains calls to inline functions. The level must be a number from zero to 255. When the level is zero, no inline expansion occurs.

579 *pointer or reference truncated by cast*

The cast expression causes a conversion of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., casting a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. Because this message indicates that some information is being lost, check the code carefully.

580 *cannot find a constructor for given initializer argument list*

The initializer list provided for the ***new*** expression does not uniquely identify a single constructor.

581 *variable '%N' can only be based on a string in this context*

All of the based modifiers can only be applied to pointer types. The only based modifier that can be applied to non-pointer types is the '__based(__segname("WATCOM"))' style.

582 *memory model modifiers are not allowed for class members*

Class members describe the arrangement and interpretation of memory and, as such, assume the memory model of the address used to access the member.

583 *redefinition of the typedef name '%S' ignored*

The compiler has detected that a slightly different type has been assigned to a typedef name. The type is functionally equivalent but typedef redefinitions should be precisely identical.

584 *constructor for variable '%S' cannot be bypassed*

The variable may not be constructed when code is executing at the position the message indicated. The C++ language places these restrictions to prevent the use of unconstructed variables.

585 *syntax error; missing start of function body after constructor initializer*

Member initializers can only be used in a constructor's definition.

Example:

```
struct S {
    int a;
    S( int x = 1 ) : a(x)
    {
    }
};
```

586 *conversion ambiguity: [expression] to [type of default argument]*

A conversion to an ambiguous base class was detected in the default argument expression.

587 *conversion of expression for default argument is impossible*

A conversion to a unrelated class was detected in the default argument expression.

588 *syntax error before template name '%s'*

The identifier in the error message has been declared as a template name at this point in the code. This may be the cause of the syntax error.

589 *private base class accessed to convert default argument*

A conversion to a private base class was detected in the default argument expression.

590 *protected base class accessed to convert default argument*

A conversion to a protected base class was detected in the default argument expression.

591 *operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

592 *left operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

593 *right operand must be an lvalue (cast produces rvalue)*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

594 *construct resolved as a declaration/type*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of a type. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

595 *construct resolved as an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of an expression (a function-like cast). The final resolution varies between compilers so it is wise to change the source code so that

the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

596 *construct cannot be resolved*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct could not be resolved by the compiler. Please report this to the Open Watcom developement team so that the problem can be analysed. See http://www.openwatcom.org/.

597 *encountered another ambiguous construct during disambiguation*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that another ambiguous construct was found inside an ambiguous construct. The compiler will correctly disambiguate the construct. The programmer is advised to change code that exhibits this warning because this is definitely uncharted territory in the C++ language.

598 *ellipsis (...) argument contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern int bar( S, ... );

static int test = bar( sv, 14, 64 );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

599 *cannot convert argument for ellipsis (...) argument*

This argument cannot be used as an ellipsis (...) argument to a function.

600 *conversion ambiguity: [argument] to [ellipsis (...) argument]*

A conversion ambiguity was detected while converting an argument to an ellipsis (...) argument.

601 *converted function type has different #pragma from original function type*

Since a #pragma can affect calling conventions, one must be very careful performing casts involving different calling conventions.

602 *class value used as return value or argument in converted function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

603 *class value used as return value or argument in original function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

604 *must look ahead to determine whether construct is a declaration/type or an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been used. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous.

605 *assembler: '%s'*

An error has been detected by the #pragma inline assembler.

606 *default argument expression cannot reference 'this'*

The order of evaluation for function arguments is unspecified in the C++ language document. Thus, a default argument must be able to be evaluated before the 'this' argument (or any other argument) is evaluated.

607 *#pragma aux must reference a "C" linkage function '%S'*

The method of assigning pragma information via the #pragma syntax is provided for compatibility with Open Watcom C. Because C only allows one function per name, this was adequate for the C language. Since C++ allows functions to be overloaded, a new method of referencing pragmas has been introduced.

Example:

```
#pragma aux this_in_SI parm caller [si] [ax];

struct S {
    void __pragma("this_in_SI") foo( int );
    void __pragma("this_in_SI") foo( char );
};
```

608 *assignment is ambiguous for operands used*

An ambiguity was detected while attempting to convert the right operand to the type of the left operand.

Example:

```
struct S1 {
    int a;
};

struct S2 : S1 {
    int b;
};

struct S3 : S2, S1 {
    int c;
};

S1* fn( S3 *p )
{
    return p;
}
```

In the example, ***class*** `S1` occurs ambiguously for an object or pointer to an object of type `S3`. A pointer to an `S3` object cannot be converted to a pointer to an `S1` object.

609 *pragma name '%s' is not defined*

Pragmas are defined with the #pragma aux syntax. See the User's Guide for the details of defining a pragma name. If the pragma has been defined then check the spelling between the definition and the reference of the pragma name.

610 *'%S' could not be generated by the compiler*

An error occurred while the compiler tried to generate the specified function. The error prevented the compiler from generating the function properly so the compilation cannot continue.

611 *'catch' does not immediately follow a 'try' or 'catch'*

The catch handler syntax must be used in conjunction with a try block.

Example:

```
void f()
{
    try {
    // code that may throw an exception
    } catch( int x ) {
    // handle 'int' exceptions
    } catch( ... ) {
    // handle all other exceptions
    }
}
```

612 *preceding catch specified '...'*

Since an ellipsis "..." catch handler will handle any type of exception, no further catch handlers can exist afterwards because they will never execute. Reorder the catch handlers so that the "..." catch handler is the last handler.

613 *argument to extern "C" function contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern "C" int bar( S );

static int test = bar( sv );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

614 *previous try block defined %L*

This informational message indicates where a preceding ***try*** block is defined.

615 *previous catch block defined %L*

This informational message indicates where a preceding ***catch*** block is defined.

616 *catch handler can never be invoked*

Because the handlers for a ***try*** block are tried in order of appearance, the type specified in a preceding ***catch*** can ensure that the current handler will never be invoked. This occurs when a base class (or reference) precedes a derived class (or reference); when a pointer to a base class (or reference to the pointer) precedes a pointer to a derived class (or reference to the pointer); or, when "void*" or "void*&" precedes a pointer or a reference to the pointer.

Example:

```
struct BASE {};
struct DERIVED : public BASE {};

foo()
{
    try {
    // code for try
    } catch( BASE b ) {     // [1]
    // code
    } catch( DERIVED ) {    // warning: [1]
    // code
    } catch( BASE* pb ) {   // [2]
    // code
    } catch( DERIVED* pd ) {// warning: [2]
    // code
    } catch( void* pv ) {   // [3]
    // code
    } catch( int* pi ) {    // warning: [3]
    // code
    } catch( BASE& br ) {   // warning: [1]
    // code
    } catch( float*& pfr ) {// warning: [3]
    // code
    }
}
```

Each erroneous catch specification indicates the preceding catch block which caused the error.

617 *cannot overload extern "C" functions (the other function is '%S')*

The C++ language only allows you to overload functions that are strictly C++ functions. The compiler will automatically generate the correct code to distinguish each particular function based on its argument types. The extern "C" linkage mechanism only allows you to define one "C" function of a particular name because the C language does not support function overloading.

618 *function will be overload ambiguous with '%S' using default arguments*

The declaration declares a function that is indistinguishable from another function of the same name with default arguments.

Example:

```
void fn( int, int = 1 );
void fn( int );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' with a default argument applied or the second 'fn' without any default arguments.

619 *linkage specification is different than previous declaration '%S'*

The linkage specification affects the binding of names throughout a program. It is important to maintain consistency because subtle problems could arise when the incorrect function is called. Usually this error prevents an unresolved symbol error during linking because the name of a declaration is affected by its linkage specification.

Example:

```
extern "C" void fn( void );
void fn( void )
{
}
```

620 *not enough segment registers available to generate '%s'*

Through a combination of options, the number of available segment registers is too small. This can occur when too many segment registers are pegged. This can be fixed by changing the command line options to only peg the segment registers that must absolutely be pegged.

621 *pure virtual destructors must have a definition*

This is an anomaly for pure virtual functions. A destructor is the only special function that is inherited and allowed to be virtual. A derived class must be able to call the base class destructor so a pure virtual destructor must be defined in a C++ program.

622 *jump into try block*

Jumps cannot enter ***try*** blocks.

Example:

```
foo( int a )
{
    if(a) goto tr_lab;

    try {
tr_lab:
    throw 1234;
    } catch( int ) {
    if(a) goto tr_lab;
    }

    if(a) goto tr_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

623 *jump into catch handler*

Jumps cannot enter ***catch*** handlers.

Example:

```
foo( int a )
{
    if(a)goto ca_lab;

    try {
    if(a)goto ca_lab;
    } catch( int ) {
ca_lab:
    }

    if(a)goto ca_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

624 *catch block does not immediately follow try block*

At least one ***catch*** handler must immediately follow the "}" of a ***try*** block.

Example:

```
extern void goop();
void foo()
{
    try {
    goop();
    }       // a catch block should follow!
}
```

In the example, there were no catch blocks after the ***try*** block.

625 *exceptions must be enabled to use feature (use 'xs' option)*

Exceptions are enabled by specifying the 'xs' option when the compiler is invoked. The error message indicates that a feature such as ***try, catch, throw,*** or function exception specification has been used without enabling exceptions.

626 *I/O error reading '%s': %s"*

When attempting to read data from a source or header file, the indicated system error occurred. Likely there is a hardware problem, or the file system has become corrupt.

627 *text following pre-processor directive*

A ***#else*** or ***#endif*** directive was found which had tokens following it rather than an end of line. Some UNIX style preprocessors allowed this, but it is not legal under standard C or C++. Make the tokens into a comment.

628 *expression is not meaningful*

This message indicates that the indicated expression is not meaningful. An expression is meaningful when a function is invoked, when an assignment or initialization is performed, or when the expression is casted to void.

Example:

```
void foo( int i, int j )
{
    i + j;  // not meaningful
}
```

629 *expression has no side effect*

The indicated expression does not cause a side effect. A side effect is caused by invoking a function, by an assignment or an initialization, or by reading a ***volatile*** variable.

Example:

```
int k;
void foo( int i, int j )
{
    i + j,  // no side effect (note comma)
    k = 3;
}
```

630 *source conversion type is '%T'*

This informational message indicates the type of the source operand, for the preceding conversion diagnostic.

631 *target conversion type is '%T'*

This informational message indicates the target type of the conversion, for the preceding conversion diagnostic.

632 *redeclaration of '%S' has different attributes*

A function cannot be made ***virtual*** or pure ***virtual*** in a subsequent declaration. All properties of a function should be described in the first declaration of a function. This is especially important for member functions because the properties of a class are affected by its member functions.

Example:

```
struct S {
    void fun();
};

virtual void S::fun()
{
}
```

633 *template class instantiation for '%T' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a class template. The final type of the template class is shown as well as the location in the source where the instantiation was initiated.

634 *template function instantiation for '%S' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a function template. The final type of the template function is shown as well as the location in the source where the instantiation was initiated.

635 *template class member instantiation was %L*

This informational message indicates that the error or warning was detected during the instantiation of a member of a class template. The location in the source where the instantiation was initiated is shown.

636 *function template binding for '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where arguments are analysed in order to infer what types should be used in a function template instantiation. The function template in question is shown along with the location in the source code that initiated the binding process.

637 *function template binding of '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where a function prototype is analysed in order to see if the prototype matches any function template of the same name. The function template in question is shown along with the location in the source code that initiated the binding process.

638 *'%s' defined %L*

This informational message indicates where the class in question was defined. The message is displayed following an error or warning diagnostic for the class in question.

Example:

```
class S;
int foo( S*p )
{
    return p->x;
}
```

The variable `p` is a pointer to an undefined class and so will cause an error to be generated. Following the error, the informational message indicates the line at which the class S was declared.

639 *form is '#pragma template_depth level' where 'level' is a non-zero number*

This ***pragma*** sets the number of times templates will be instantiated for nested instantiations. The depth check prevents infinite compile times for incorrect programs.

640 *possible non-terminating template instantiation (use "#pragma template_depth %d" to increase depth)*

This message indicates that a large number of expansions were required to complete a template class or template function instantiation. This may indicate that there is an erroneous use of a template. If the program will complete given more depth, try using the suggested #pragma in the error message to increase the depth. The number provided is double the previous value.

641 *cannot inherit a partially defined base class '%T'*

This message indicates that the base class was in the midst of being defined when it was inherited. The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:

```
struct Partial {
    struct Nested : Partial {
    int n;
    };
};
```

642 *ambiguous function: %F defined %L*

This informational message shows the functions that were detected to be ambiguous.

Example:

```
int amb( char );         // will be ambiguous
int amb( unsigned char );   // will be ambiguous
int amb( char, char );
int k = amb( 14 );
```

The constant value 14 has an ***int*** type and so the attempt to invoke the function `amb` is ambiguous. The first two functions are ambiguous (and will be displayed); the third is not considered (nor displayed) since it is declared to have a different number of arguments.

643 *cannot convert argument %d defined %L*

This informational message indicates the first argument which could not be converted to the corresponding type for the declared function. It is displayed when there is exactly one function declared with the indicated name.

644 *'this' cannot be converted*

This informational message indicates the ***this*** pointer for the function which could not be converted to the type of the ***this*** pointer for the declared function. It is displayed when there is exactly one function declared with the indicated name.

645 *rejected function: %F defined %L*

This informational message shows the overloaded functions which were rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function with the indicated name.

646 *'%T' operator can be used*

Following a diagnosis of operator ambiguity, this information message indicates that the operator can be applied with operands of the type indicated in the message.

Example:

```
struct S {
    S( int );
    operator int();
    S operator+( int );
};
S s(15);
int k = s + 123;    // "+" is ambiguous
```

In the example, the "+" operation is ambiguous because it can implemented as by the addition of two integers (with `S::operator int` applied to the second operand) or by a call to `S::operator+`. This informational message indicates that the first is possible.

647 *cannot #undef '%s'*

The predefined macros `__cplusplus`, `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, `__TIME__`, `__FUNCTION__` and `__func__` cannot be undefined using the ***#undef*** directive.

Example:

```
#undef __cplusplus
#undef __DATE__
#undef __FILE__
#undef __LINE__
#undef __STDC__
#undef __TIME__
#undef __FUNCTION__
#undef __func__
```

All of the preceding directives are not permitted.

648 *cannot #define '%s'*

The predefined macros `__cplusplus`, `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, and `__TIME__` cannot be defined using the ***#define*** directive.

Example:

```
#define __cplusplus     1
#define __DATE__        2
#define __FILE__        3
#define __LINE__        4
#define __STDC__        5
#define __TIME__        6
```

All of the preceding directives are not permitted.

649 *template function '%F' defined %L*

This informational message indicates where the function template in question was defined. The message is displayed following an error or warning diagnostic for the function template in question.

Example:

```
template <class T>
    void foo( T, T * )
    {
    }

void bar()
{
    foo(1);     // could not instantiate
}
```

The function template for `foo` cannot be instantiated for a single argument causing an error to be generated. Following the error, the informational message indicates the line at which `foo` was declared.

650 *ambiguous function template: %F defined %L*

This informational message shows the function templates that were detected to be ambiguous for the arguments at the call point.

651 *cannot instantiate %S*

This message indicates that the function template could not be instantiated for the arguments supplied. It is displayed when there is exactly one function template declared with the indicated name.

652 *rejected function template: %F defined %L*

This informational message shows the overloaded function template which was rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function or function template with the indicated name.

653 *operand cannot be a function*

The indicated operation cannot be applied to a function.

Example:

```
int Fun();
int j = ++Fun;  // illegal
```

In the example, the attempt to increment a function is illegal.

654 *left operand cannot be a function*

The indicated operation cannot be applied to the left operand which is a function.

Example:

```
extern int Fun();
void foo()
{
    Fun = 0;    // illegal
}
```

In the example, the attempt to assign zero to a function is illegal.

655 *right operand cannot be a function*

The indicated operation cannot be applied to the right operand which is a function.

Example:

```
extern int Fun();
void foo()
{
    void* p = 3[Fun];   // illegal
}
```

In the example, the attempt to subscript a function is illegal.

656 *define this function inside its class definition (may improve code quality)*

The Open Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations.

Example:

```
struct S {
    ~S();
};

S::~S() {
}
```

657 *define this function inside its class definition (could have improved code quality)*

The Open Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations. This particular warning indicates that the compiler has already found an opportunity in previous code but it found out too late that the constructor or destructor had an empty function body.

Example:

```
struct S {
    ~S();
};
struct T : S {
    ~T() {}
};

S::~S() {
}
```

658 *cannot convert address of overloaded function '%S'*

This information message indicates that an address of an overloaded function cannot be converted to the indicated type.

Example:

```
int ovload( char );
int ovload( float );
int routine( int (*)( int );
int k = routine( ovload );
```

The first argument for the function `routine` cannot be converted, resulting in the informational message.

659 *expression cannot have void type*

The indicated expression cannot have a ***void*** type.

Example:

```
main( int argc, char* argv )
{
    if( (void)argc ) {
    return 5;
    } else {
    return 9;
    }
}
```

Conditional expressions, such as the one illustrated in the ***if*** statement cannot have a ***void*** type.

660 *cannot reference a bit field*

The smallest addressable unit is a byte. You cannot reference a bit field.

Example:
```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
int& ref = var.bitfield;    // illegal
```

661 *cannot assign to object having an undefined class*

An assignment cannot be be made to an object whose class has not been defined.

Example:
```
class X;        // declared, but not defined
extern X& foo();    // returns reference (ok)
extern X obj;
void goop()
{
    obj = foo();    // error
}
```

662 *cannot create member pointer to constructor*

A member pointer value cannot reference a constructor.

Example:
```
class C {
    C();
};
int foo()
{
    return 0 == &C::C;
}
```

663 *cannot create member pointer to destructor*

A member pointer value cannot reference a destructor.

Example:
```
class C {
    ~C();
};
int foo()
{
    return 0 == &C::~C;
}
```

664 *attempt to initialize a non-constant reference with a temporary object*

A temporary value cannot be converted to a non-constant reference type.

Example:

```
struct C {
    C( C& );
    C( int );
};

C & c1 = 1;
C c2 = 2;
```

The initializations of `c1` and `c2` are erroneous, since temporaries are being bound to non-const references. In the case of `c1,` an implicit constructor call is required to convert the integer to the correct object type. This results in a temporary object being created to initialize the reference. Subsequent code can modify this temporary's state. The initialization of `c2,` is erroneous for a similar reason. In this case, the temporary is being bound to the non-const reference argument of the copy constructor.

665 *temporary object used to initialize a non-constant reference*

Ordinarily, a temporary value cannot be bound to a non-constant reference. There is enough legacy code present that the Open Watcom C++ compiler issues a warning in cases that should be errors. This may change in the future so it is advisable to correct the code as soon as possible.

666 *assuming unary 'operator &' not overloaded for type '%T'*

An explicit address operator can be applied to a reference to an undefined class. The Open Watcom C++ compiler will assume that the address is required but it does not know whether this was the programmer's intention because the class definition has not been seen.

Example:

```
struct S;

S * fn( S &y ) {
    // assuming no operator '&' defined
    return &y;
}
```

667 *'va_start' macro will not work without an argument before '...'*

The warning indicates that it is impossible to access the arguments passed to the function without declaring an argument before the "..." argument. The "..." style of argument list (without any other arguments) is only useful as a prototype or if the function is designed to ignore all of its arguments.

Example:

```
void fn( ... )
{
}
```

668 *'va_start' macro will not work with a reference argument before '...'*

The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the reference argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

void fn( int &r, ... )
{
    va_list args;

    // address of 'r' is address of
    // object 'r' references so
    // 'va_start' will not work properly
    va_start( args, r );
    va_end( args );
}
```

669 *'va_start' macro will not work with a class argument before '...'*

This warning is specific to C++ compilers that quietly convert class arguments to class reference arguments. The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the class argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

struct S {
    S();
};

void fn( S c, ... )
{
    va_list args;

    // Open Watcom C++ passes a pointer to
    // the temporary created for passing
    // 'c' rather than pushing 'c' on the
    // stack so 'va_start' will not work
    // properly
    va_start( args, c );
    va_end( args );
}
```

670 *function modifier conflicts with previous declaration '%S'*

The symbol declaration conflicts with a previous declaration with regard to function modifiers. Either the previous declaration did not have a function modifier or it had a different one.

Example:

```
#pragma aux never_returns aborts;

void fn( int, int );
void __pragma("never_returns") fn( int, int );
```

671 *function modifier cannot be used on a variable*

The symbol declaration has a function modifier being applied to a variable or non-function. The cause of this may be a declaration with a missing function argument list.

Example:

```
int (* __pascal ok)();
int (* __pascal not_ok);
```

672 *'%T' contains the following pure virtual functions*

This informational message indicates that the class contains pure virtual function declarations. The class is definitely abstract as a result and cannot be used to declare variables. The pure virtual functions declared in the class are displayed immediately following this message.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};

A x;
```

673 *'%T' has no implementation for the following pure virtual functions*

This informational message indicates that the class is derived from an abstract class but the class did not override enough virtual function declarations. The pure virtual functions declared in the class are displayed immediately following this message.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

674 *pure virtual function '%F' defined %L*

This informational message indicates that the pure virtual function has not been overridden. This means that the class is abstract.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

675 *restriction: standard calling convention required for '%S'*

The indicated function may be called by the C++ run-time system using the standard calling convention. The calling convention specified for the function is incompatible with the standard convention. This message may result when `__pascal` is specified for a default constructor, a copy constructor, or a destructor. It may also result when `parm reverse` is specified in a ***#pragma*** for the function.

676 *number of arguments in function call is incorrect*

The number of arguments in the function call does not match the number declared for the function type.

Example:

```
extern int (*pfn)( int, int );
int k = pfn( 1, 2, 3 );
```

In the example, the function pointer was declared to have two arguments. Three arguments were used in the call.

677 *function has type '%T'*

This informational message indicates the type of the function being called.

678 *invalid octal constant*

The constant started with a '0' digit which makes it look like an octal constant but the constant contained the digits '8' and '9'. The problem could be an incorrect octal constant or a missing '.' for a floating constant.

Example:

```
int i = 0123456789; // invalid octal constant
double d = 0123456789;  // missing '.'?
```

679 *class template definition started %L*

This informational message indicates where the class template definition started so that any problems with missing braces can be fixed quickly and easily.

Example:

```
template <class T>
    struct S {
    void f1() {
    // error missing '}'
    };

template <class T>
    struct X {
    void f2() {
    }
    };
```

680 *constructor initializer started %L*

This informational message indicates where the constructor initializer started so that any problems with missing parenthesis can be fixed quickly and easily.

Example:

```
struct S {
    S( int x ) : a(x), b(x // missing parenthesis
    {
    }
};
```

681 *zero size array must be the last data member*

The language extension that allows a zero size array to be declared in a class definition requires that the array be the last data member in the class.

Example:

```
struct S {
    char a[];
    int b;
};
```

682 *cannot inherit a class that contains a zero size array*

The language extension that allows a zero size array to be declared in a class definition disallows the use of the class as a base class. This prevents the programmer from corrupting storage in derived classes through the use of the zero size array.

Example:

```
struct B {
    int b;
    char a[];
};
struct D : B {
    int d;
};
```

683 *zero size array '%S' cannot be used in a class with base classes*

The language extension that allows a zero size array to be declared in a class definition requires that the class not have any base classes. This is required because the C++ compiler must be free to organize base classes in any manner for optimization purposes.

Example:

```
struct B {
    int b;
};
struct D : B {
    int d;
    char a[];
};
```

684 *cannot catch abstract class object*

C++ does not allow abstract classes to be instantiated and so an abstract class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an abstract class.

Example:

```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
    throw Derived();
    } catch( Abstract abstract ) {   // object
    xyz = 1;
    }
}
```

The catch clause in the preceding example would be diagnosed as improper, since an abstract class is specified. The example could be coded as follows.

Example:

```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
    throw Derived();
    } catch( Abstract & abstract ) {  // reference
    xyz = 1;
    }
}
```

685 *non-static member function '%S' cannot be specified*

The indicated non-static member function cannot be used in this context. For example, such a function cannot be used as the second or third operand of the conditional operator.

Example:

```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return (i ? foo : bar)();
}
```

Neither `foo` nor `bar` can be specified as shown in the example. The example can be properly coded as follows:

Example:

```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return i ? foo() : bar();
}
```

686 *attempt to convert pointer or reference from a base to a derived class*

A pointer or reference to a base class cannot be converted to a pointer or reference, respectively, of a derived class, unless there is an explicit cast. The `return` statements in the following example will be diagnosed.

Example:
```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return &b; }
Derived& ReturnRef() { return b; }
```

The following program would be acceptable:

Example:
```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return (Derived*)&b; }
Derived& ReturnRef() { return (Derived&)b; }
```

687 *expression for 'while' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ;)` without causing warnings.

688 *testing expression for 'for' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ;)` without causing warnings.

689 *conditional expression is always true (non-zero)*

The indicated expression is a non-zero constant and so will always be true.

690 *conditional expression is always false (zero)*

The indicated expression is a zero constant and so will always be false.

691 *expecting a member of '%T' to be defined in this context*

A class template member definition must define a member of the associated class template. The complexity of the C++ declaration syntax can make this error hard to identify visually.

Example:

```
template <class T>
    struct S {
    typedef int X;
    static X fn( int );
    static X qq;
    };

template <class T>
    S<T>::X fn( int ) {// should be 'S<T>::fn'

    return fn( 2 );
    }

template <class T>
    S<T>::X qq = 1; // should be 'S<T>::q'

S<int> x;
```

692 *cannot throw an abstract class*

An abstract class cannot be thrown since copies of that object may have to be made (which is impossible);

Example:

```
struct abstract_class {
    abstract_class( int );
    virtual int foo() = 0;
};

void goop()
{
    throw abstract_class( 17 );
}
```

The ***throw*** expression is illegal since it specifies an abstract class.

693 *cannot create pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for write access.

694 *error occurred while writing pre-compiled header file*

The compiler has detected a problem while trying to write some data to the pre-compiled header file.

695 *error occurred while reading pre-compiled header file*

The compiler has detected a problem while trying to read some data from the pre-compiled header file.

696 *pre-compiled header file being recreated*

The existing pre-compiled header file may either be corrupted or is a version that the compiler cannot use due to updates to the compiler. A new version of the pre-compiled header file will be created.

697 *pre-compiled header file being recreated (different compile options)*

The compiler has detected that the command line options have changed enough so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

698 *pre-compiled header file being recreated (different #include file)*

The compiler has detected that the first ***#include*** file name is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

699 *pre-compiled header file being recreated (different current directory)*

The compiler has detected that the working directory is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

700 *pre-compiled header file being recreated (different INCLUDE path)*

The compiler has detected that the INCLUDE path is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

701 *pre-compiled header file being recreated ('%s' has been modified)*

The compiler has detected that an include file has changed so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

702 *pre-compiled header file being recreated (macro '%s' is different)*

The compiler has detected that a macro definition is different so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

703 *pre-compiled header file being recreated (macro '%s' is not defined)*

The compiler has detected that a macro has not been defined so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

704 *command line specifies smart windows callbacks and DS not equal to SS*

An illegal combination of switches has been detected. The windows smart callbacks option cannot be combined with either of the build DLL or DS not equal to SS options.

705 *class '%N' cannot be used with #pragma dump_object_model*

The indicated name has not yet been declared or has been declared but not yet been defined as a class. Consequently, the object model cannot be dumped.

706 *repeated modifier is '%s'*

This informational message indicates what modifier was repeated in the declaration.

Example:

```
typedef int __far FARINT;
FARINT __far *p;    // repeated __far modifier
```

707 *semicolon (';') may be missing after class/enum definition*

This informational message indicates that a missing semicolon (';') may be the cause of the error.

Example:

```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

708 *cannot return a type of unknown size*

A value of an unknown type cannot be returned.

Example:

```
class S;
S foo();

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

In the example, foo cannot be invoked because the class which it returns has not been defined.

709 *cannot initialize array member '%S'*

An array class member cannot be specified as a constructor initializer.

Example:
```
class S {
public:
    int arr[3];
    S();
};
S::S() : arr( 1, 2, 3 ) {}
```

In the example, `arr` cannot be specified as a constructor initializer. Instead, the array may be initialized within the body of the constructor.

Example:
```
class S {
public:
    int arr[3];
    S();
};
S::S()
{
    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;
}
```

710 *file '%s' will #include itself forever*

The compiler has detected that the file in the message has been ***#include*** from within itself without protecting against infinite inclusion. This can happen if ***#ifndef*** and ***#define*** header file protection has not been used properly.

Example:
```
#include __FILE__
```

711 *'mutable' may only be used for non-static class members*

A declaration in file scope or block scope cannot have a storage class of ***mutable.***

Example:
```
mutable int a;
```

712 *'mutable' member cannot also be const*

A ***mutable*** member can be modified even if its class object is ***const.*** Due to the semantics of ***mutable,*** the programmer must decide whether a member will be ***const*** or ***mutable*** because it cannot be both at the same time.

Example:

```
struct S {
    mutable const int * p;  // OK
    mutable int * const q;  // error
};
```

713 *left operand cannot be of type bool*

The left hand side of an assignment operator cannot be of type ***bool*** except for simple assignment. This is a restriction required in the C++ language.

Example:

```
bool q;

void fn()
{
    q += 1;
}
```

714 *operand cannot be of type bool*

The operand of both postfix and prefix "--" operators cannot be of type ***bool.*** This is a restriction required in the C++ language.

Example:

```
bool q;

void fn()
{
    --q;   // error
    q--;   // error
}
```

715 *member '%N' has not been declared in '%T'*

The compiler has found a member which has not been previously declared. The symbol may be spelled differently than the declaration, or the declaration may simply not be present.

Example:

```
struct X { int m; };

void fn( X *p )
{
    p->x = 1;
}
```

716 *integral value may be truncated*

This message indicates that the compiler knows that all values will not be preserved after the assignment or initialization. If this is acceptable, cast the value to the appropriate type in the assignment or initialization.

Example:

```
char inc( char c )
{
    return c + 1;
}
```

717 *left operand type is '%T'*

This informational message indicates the type of the left hand side of the expression.

718 *right operand type is '%T'*

This informational message indicates the type of the right hand side of the expression.

719 *operand type is '%T'*

This informational message indicates the type of the operand.

720 *expression type is '%T'*

This informational message indicates the type of the expression.

721 *virtual function '%S' cannot have its return type changed*

This restriction is due to the relatively new feature in the C++ language that allows return values to be changed when a virtual function has been overridden. It is not possible to support both features because in order to support changing the return value of a function, the compiler must construct a "wrapper" function that will call the virtual function first and then change the return value and return. It is not possible to do this with "..." style functions because the number of parameters is not known.

Example:

```
struct B {
};
struct D : virtual B {
};

struct X {
    virtual B *fn( int, ... );
};
struct Y : X {
    virtual D *fn( int, ... );
};
```

722 *__declspec('%N') is not supported*

The identifier used in the ***__declspec*** declaration modifier is not supported by Open Watcom C++.

723 *attempt to construct a far object when the data model is near*

Constructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

724 *-zo is an obsolete switch (has no effect)*

The ***-zo*** option was required in an earlier version of the compiler but is no longer used.

725 *"%s"*

This is a user message generated with the ***#pragma message*** preprocessing directive.

Example:

```
#pragma message( "my very own warning" );
```

726 *no reference to formal parameter '%S'*

There are no references to the declared formal parameter. The simplest way to remove this warning in C++ is to remove the name from the argument declaration.

Example:

```
int fn1( int a, int b, int c )
{
    // 'b' not referenced
    return a + c;
}
int fn2( int a, int /* b */, int c )
{
    return a + c;
}
```

727 *cannot dereference a pointer to void*

A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

Example:

```
void fn( void *p )
{
    return *p;
}
```

728 *class modifiers for '%T' conflict with class modifiers for '%T'*

A conflict between class modifiers for classes related through inheritance has been detected. A conflict will occur if two base classes have class modifiers that are different. The conflict can be resolved by ensuring that all classes related through inheritance have the same class modifiers. The default resolution is to have no class modifier for the derived base.

Example:

```
struct __cdecl B1 {
    void fn( int );
};
struct __stdcall B2 {
    void fn( int );
};
struct D : B1, B2 {
};
```

729 *invalid hexadecimal constant*

The constant started with a '0x' prefix which makes it look like a hexadecimal constant but the constant was not followed by any hexadecimal digits.

Example:

```
unsigned i = 0x;      // invalid hex constant
```

730 *return type of 'operator ->' will not allow '->' to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.*** This warning indicates that the ***operator ->*** can never be used as an overloaded operator. The only way the operator can be used is to explicitly call it by name.

Example:

```
struct S {
    int a;
    void *operator ->();
};

void *fn( S &q )
{
    return q.operator ->();
}
```

731 *class should have a name since it needs a constructor or a destructor*

The class definition does not have a class name but it includes members that have constructors or destructors. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:

```
struct P {
    int x,y;
    P();
};

typedef struct {
    P c;
    int v;
} T;
```

732 *class should have a name since it inherits a class*

The class definition does not have a class name but it inherits a class. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:

```
struct P {
    int x,y;
    P();
};

typedef struct : P {
    int v;
} T;
```

733 *cannot open pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for read/write access.

734 *invalid second argument to va_start*

The second argument to the va_start macro should be the name of the argument just before the "..." in the argument list.

735 *'//' style comment continues on next line*

The compiler has detected a line continuation during the processing of a C++ style comment ("//"). The warning can be removed by switching to a C style comment ("/**/"). If you require the comment to be terminated at the end of the line, make sure that the backslash character is not the last character in the line.

Example:

```
#define XX 23 // comment start \
comment \
end

int x = XX; // comment start ...\
comment end
```

736 *cannot open file '%s' for write access*

The compiler has detected a problem while trying to open the indicated file for write access.

737 *implicit conversion of pointers to integral types of same size*

According to the ISO/ANSI Draft Working Paper, a string literal is an array of ***char.*** Consequently, it is illegal to initialize or assign the pointer resulting from that literal to a pointer of either ***unsigned char*** or ***signed char,*** since these pointers point at objects of a different type.

738 *option requires a number*

The specified option is not recognized by the compiler since there was no number after it (i.e., "-w=1"). Numbers must be non-negative decimal numbers.

739 *option -fc specified more than once*

The -fc option can be specified at most once on a command line.

740 *option -fc specified in batch file of commands*

The -fc option cannot be specified on a line in the batch file of command lines specified by the -fc option on the command line used to invoke the compiler.

741 *file specified by -fc is empty or cannot be read*

The file specified using the -fc option is either empty or an input/output error was diagnosed for the file.

742 *cannot open file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

743 *input/output error reading the file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

744 *'%N' does not have a return type specified (int assumed)*

In C++, operator functions should have an explicit return type specified. In future revisions of the ISO/ANSI C++ standard, the use of default int type specifiers may be prohibited so removing any dependencies on default int early will prevent problems in the future.

Example:

```
struct S {
    operator = ( S const & );
    operator += ( S const & );
};
```

745 *cannot initialize reference to non-constant with a constant object*

A reference to a non-constant object cannot be initialized with a reference to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:

```
extern const int *pic;
extern int & ref = pic;
```

746 *processing %s*

This informational message indicates where an error or warning was detected while processing the switches specified on the command line, in environment variables, in command files (using the '@' notation), or in the batch command file (specified using the -fc option).

747 *class '%T' has not been defined*

This informational message indicates a class which was not defined. This is noted following an error or warning message because it often helps to a user to determine the cause of that diagnostic.

748 *cannot catch undefined class object*

C++ does not allow abstract classes to be copied and so an undefined class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an undefined class.

749 *class '%T' cannot be used since its definition has errors*

The analysis of the expression could not continue due to previous errors diagnosed in the class definition.

750 *function prototype in block scope missing 'extern'*

This warning can be triggered when the intent is to define a variable with a constructor. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as a function prototype. A work-around for this problem is contained in the example. If a prototype is desired, add the ***extern*** storage class to remove this warning.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    S a( C() ); // function prototype!
    S b( (C()) );// variable definition

    int bar( int );// warning
    extern int sam( int ); // no warning
}
```

751 *function prototype is '%T'*

This informational message indicates what the type of the function prototype is for the message in question.

752 *class '%T' contains a zero size array*

This warning is triggered when a class with a zero sized array is used in an array or as a class member. This is a questionable practice since a zero sized array at the end of a class often indicates a class that is dynamically sized when it is constructed.

Example:

```
struct C {
    C *next;
    char name[];
};

struct X {
    C q;
};

C a[10];
```

753 *invalid 'new' modifier*

The Open Watcom C++ compiler does not support new expression modifiers but allows them to match the ambient memory model for compatibility. Invalid memory model modifiers are also rejected by the compiler.

Example:

```
int *fn( unsigned x )
{
    return new __interrupt int[x];
}
```

754 *'__declspec(thread)' data '%S' must be link-time initialized*

This error message indicates that the data item in question either requires a constructor, destructor, or run-time initialization. This cannot be supported for thread-specific data at this time.

Example:

```
#include <stdlib.h>

struct C {
    C();
};
struct D {
    ~D();
};

C __declspec(thread) c;
D __declspec(thread) d;
int __declspec(thread) e = rand();
```

755 *code may not work properly if this module is split across a code segment*

The "zm" option allows the compiler to generate functions into separate segments that have different names so that more than 64k of code can be generated in one object file. Unfortunately, if an explicit near function is coded in a large code model, the possibility exists that the linker can place the near function in a separate code segment than a function that calls it. This would cause a linker error followed by an execution error if the executable is executed. The "zmf" option can be used if you require explicit near functions in your code.

Example:

```
// These functions may not end up in the
// same code segment if the -zm option
// is used. If this is the case, the near
// call will not work since near functions
// must be in the same code segment to
// execute properly.
static int near near_fn( int x )
{
    return x + 1;
}

int far_fn( int y )
{
    return near_fn( y * 2 );
}
```

756 *#pragma extref: symbol '%N' not declared*

This error message indicates that the symbol referenced by ***#pragma extref*** has not been declared in the context where the pragma was encountered.

757 *#pragma extref: overloaded function '%S' cannot be used*

An external reference can be emitted only for external functions which are not overloaded.

758 *#pragma extref: '%N' is not a function or data*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the referenced symbol is neither a function nor a data item. An external reference can be emitted only for external functions which are not overloaded and for external data items.

759 *#pragma extref: '%S' is not external*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the symbol is not external. An external reference can be emitted only for external functions which are not overloaded and for external data items.

760 *pre-compiled header file being recreated (debugging info may change)*

The compiler has detected that the module being compiled was used to create debugging information for use by other modules. In order to maintain correctness, the pre-compiled header file must be recreated along with the object file.

761 *octal escape sequence out of range; truncated*

This message indicates that the octal escape sequence produces an integer that cannot fit into the required character type.

Example:

```
char *p = "\406";
```

762 *binary operator '%s' missing right operand*

There is no expression to the right of the indicated binary operator.

763 *binary operator '%s' missing left operand*

There is no expression to the left of the indicated binary operator.

764 *expression contains extra operand(s)*

The expression contains operand(s) without an operator

765 *expression contains consecutive operand(s)*

More than one operand found in a row.

766 *unmatched right parenthesis ')'*

The expression contains a right parenthesis ")" without a matching left parenthesis.

767 *unmatched left parenthesis '('*

The expression contains a left parenthesis "(" without a matching right parenthesis.

768 *no expression between parentheses '( )'*

There is a matching set of parenthesis "()" which do not contain an expression.

769 *expecting ':' operator in conditional expression*

A conditional expression exists without the ':' operator.

770 *expecting '?' operator in conditional expression*

A conditional expression exists without the '?' operator.

771 *expecting first operand in conditional expression*

A conditional expression exists without the first operand.

772 *expecting second operand in conditional expression*

A conditional expression exists without the second operand.

773 *expecting third operand in conditional expression*

A conditional expression exists without the third operand.

774 *expecting operand after unary operator '%s'*

A unary operator without being followed by an operand.

775 *'%s' unexpected in constant expression*

'%s' not allowed in constant expression

776 *assembler: '%s'*

A warning has been issued by the #pragma inline assembler.

777 *expecting 'id' after '::' but found '%s'*

The '::' operator has an invalid token following it.

Example:

```
#define fn( x ) ((x)+1)

struct S {
    int inc( int y ) {
    return ::fn( y );
    }
};
```

778 *only constructors can be declared explicit*

Currently, only constructors can be declared with the ***explicit*** keyword.

Example:

```
int explicit fn( int x ) {
    return x + 1;
}
```

779 *const_cast type must be pointer, member pointer, or reference*

The type specified in a ***const_cast*** operator must be a pointer, a pointer to a member of a class, or a reference.

Example:

```
extern int const *p;
long lp = const_cast<long>( p );
```

780 *const_cast expression must be pointer to same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to the same type of object as that specified in the ***const_cast*** operator.

Example:

```
extern int const * ip;
long* lp = const_cast<long*>( ip );
```

781 *const_cast expression must be lvalue of the same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be an lvalue or reference to the same type of object as that specified in the ***const_cast*** operator.

Example:

```
extern int const i;
long& lr = const_cast<long&>( i );
```

782 *expression must be pointer to member from same class in const_cast*

The expression must be a pointer to member from the same class as that specified in the ***const_cast*** operator.

Example:

```
struct B {
    int ib;
};
struct D : public B {
};
extern int const B::* imb;
int D::* imd const_cast<int D::*>( imb );
```

783 *expression must be member pointer to same type as specified in const_cast*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to member of the same type as that specified in the ***const_cast*** operator.

Example:

```
struct B {
    int ib;
    long lb;
};
int D::* imd const_cast<int D::*>( &B::lb );
```

784 *reinterpret_cast expression must be pointer or integral object*

When a pointer type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer or an integer.

Example:

```
extern float fval;
long* lp = const_cast<long*>( fval );
```

The expression has ***float*** type and so is illegal.

785 *reinterpret_cast expression cannot be casted to reference type*

When a reference type is specified in the ***reinterpret_cast*** operator, the expression must be an lvalue (or have reference type). Additionally, constness cannot be casted away.

Example:

```
extern long f;
extern const long f2;
long& lr1 = const_cast<long&>( f + 2 );
long& lr2 = const_cast<long&>( f2 );
```

Both initializations are illegal. The first cast expression is not an lvalue. The second cast expression attempts to cast away constness.

786 *reinterpret_cast expression cannot be casted to pointer to member*

When a pointer to member type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer to member. Additionally, constness cannot be casted away.

Example:

```
extern long f;
struct S {
    const long f2;
    S();
};
long S::* mp1 = const_cast<long S:: *>( f );
long S::* mp2 = const_cast<long S:: *>( &S::f2 );
```

Both initializations are illegal. The first cast expression does not involve a member pointer. The second cast expression attempts to cast away constness.

787 *only integral arithmetic types can be used with reinterpret_cast*

Pointers can only be casted to sufficiently large integral types.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
```

The cast is illegal because ***float*** type is specified.

788 *only integral arithmetic types can be used with reinterpret_cast*

Only integral arithmetic types can be casted to pointer types.

Example:

```
float flt;
void* p = reinterpret_cast<void*>( flt );
```

The cast is illegal because `flt` has ***float*** type which is not integral.

789 *cannot cast away constness*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S const * ps;
extern int const S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness.

790 *size of integral type in cast less than size of pointer*

An object of the indicated integral type is too small to contain the value of the indicated pointer.

Example:

```
int x;
char p = reinterpret_cast<char>( &x );
char q = (char)( &x );
```

Both casts are illegal since a ***char*** is smaller than a pointer.

791 *type cannot be used in reinterpret_cast*

The type specified with reinterpret_cast must be an integral type, a pointer type, a pointer to a member of a class, or a reference type.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

792 *only pointers can be casted to integral types with reinterpret_cast*

The expression must be a pointer type.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

793 *only integers and pointers can be casted to pointer types with reinterpret_cast*

The expression must be a pointer or integral type.

Example:

```
void* x;
void* p = reinterpret_cast<void*>( 16 );
void* q = ( reinterpret_cast<void*>( x ), p );
```

The casts specify illegal types.

794 *static_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***static_cast*** operator. Perhaps reinterpret_cast or dynamic_cast should be used instead;

795 *static_cast cannot be used with the type specified*

A static cast cannot be used with a function type or array type.

Example:

```
typedef int fun( int );
extern int poo( long );
int i = ( static_cast<fun)( poo ) )( 22 );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

796 *static_cast cannot be used with the reference type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
long lng;
int& ref = static_cast<int&>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

797 *static_cast cannot be used with the pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
long lng;
int* ref = static_cast<int*>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

798 *static_cast cannot be used with the member pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
struct S {
    long lng;
};
int S::* mp = static_cast<int S::*>( &S::lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

799 *static_cast type is ambiguous*

More than one constructor and/or used-defined conversion function can be used to convert the expression to the indicated type.

800 *cannot cast from ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the base class to the derived class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Derived* foo( Base* p )
{
    return static_cast<Derived*>( p );
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

801 *cannot cast to ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the derived class to the base class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Base* foo( Derived* p )
{
    return (Base*)p;
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

802 *can only static_cast integers to enumeration type*

When an enumeration type is specified with ***static_cast,*** the expression must be an integer.

Example:

```
enum sex { male, female };
sex father = static_cast<sex>( 1.0 );
```

The cast is illegal because the expression is not an integer.

803 *dynamic_cast cannot be used with the type specified*

A dynamic cast can only specify a reference to a class or a pointer to a class or ***void.*** When a class is referenced, it must have virtual functions defined within that class or a base class of that class.

804 *dynamic_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***dynamic_cast*** operator. Only a pointer or reference to a class object can be converted. When a class object is referenced, it must have virtual functions defined within that class or a base class of that class.

805 *dynamic_cast requires class '%T' to have virtual functions*

The indicated class must have virtual functions defined within that class or a base class of that class.

806 *base class for type in dynamic_cast is ambiguous (will fail)*

The type in the ***dynamic_cast*** is a pointer or reference to an ambiguous base class.

Example:

```
struct A { virtual void f(){}; };
struct D1 : A { };
struct D2 : A { };
struct D : D1, D2 { };

A *foo( D *p ) {
    // will always return NULL
    return( dynamic_cast< A* >( p ) );
}
```

807 *base class for type in dynamic_cast is private (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a private base class.

Example:

```
struct V { virtual void f(){}; };
struct A : private virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

808 *base class for type in dynamic_cast is protected (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a protected base class.

Example:

```
struct V { virtual void f(){}; };
struct A : protected virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

809 *type cannot be used with an explicit cast*

The indicated type cannot be specified as the type of an explicit cast. For example, it is illegal to cast to an array or function type.

810 *cannot cast to an array type*

It is not permitted to cast to an array type.

Example:

```
typedef int array_type[5];
int array[5];
int* p = (array_type)array;
```

811 *cannot cast to a function type*

It is not permitted to cast to a function type.

Example:

```
typedef int fun_type( void );
void* p = (fun_type)0;
```

812 *implementation restriction: cannot generate RTTI info for '%T' (%d classes)*

The information for one class must fit into one segment. If the segment size is restricted to 64k, the compiler may not be able to emit the correct information properly if it requires more than 64k of memory to represent the class hierarchy.

813 *more than one default constructor for '%T'*

The compiler found more than one default constructor signature in the class definition. There must be only one constructor declared that accepts no arguments.

Example:

```
struct C {
    C();
    C( int = 0 );
};
C cv;
```

814 *user-defined conversion is ambiguous*

The compiler found more than one user-defined conversion which could be performed. The indicated functions that could be used are shown.

Example:

```
struct T {
    T( S const& );
};
struct S {
    operator T const& ();
};
extern S sv;
T const & tref = sv;
```

Either the constructor or the conversion function could be used; consequently, the conversion is ambiguous.

815 *range of possible values for type '%T' is %s to %s*

This informational message indicates the range of values possible for the indicated unsigned type.

Example:

```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the possible range of values for the unsigned type involved.

816 *range of possible values for type '%T' is %s to %s*

This informational message indicates the range of values possible for the indicated signed type.

Example:

```
signed char c;
if( c <= 127 );
```

Because the value of signed char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the possible range of values for the signed type involved.

817 *constant expression in comparison has value %s*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:

```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the constant value (0 in this case) involved in the comparison.

818 *constant expression in comparison has value %s*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:

```
signed char c;
if( c <= 127 );
```

Because the value of char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the constant value (127 in this case) involved in the comparison.

819 *conversion of const reference to non-const reference*

A reference to a constant object is being converted to a reference to a non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int const & const_ref;
int & non_const_ref = const_ref;
```

820 *conversion of volatile reference to non-volatile reference*

A reference to a volatile object is being converted to a reference to a non-volatile object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int volatile & volatile_ref;
int & non_volatile_ref = volatile_ref;
```

821 *conversion of const volatile reference to plain reference*

A reference to a constant and volatile object is being converted to a reference to a non-volatile and non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int const volatile & const_volatile_ref;
int & non_const_volatile_ref = const_volatile_ref;
```

822 *current declaration has type '%T'*

This informational message indicates the type of the current declaration that caused the message to be issued.

Example:

```
extern int __near foo( int );
extern int __far foo( int );
```

823 *only a non-volatile const reference can be bound to temporary*

The expression being bound to a reference will need to be converted to a temporary of the type referenced. This means that the reference will be bound to that temporary and so the reference must be a non-volatile const reference.

Example:

```
extern int * pi;
void * & r1 = pi;       // error
void * const & r2 = pi;     // ok
void * volatile & r3 = pi;  // error
void * const volatile & r4 = pi;// error
```

824 *conversion of pointer to member across a virtual base*

In November 1995, the Draft Working Paper was amended to disallow pointer to member conversions when the source class is a virtual base of the target class. This situation is treated as a warning (unless -za is specified to require strict conformance), as a temporary measure. In the future, an error will be diagnosed for this situation.

Example:

```
struct B {
    int b;
};

struct D : virtual B {
    int d;
};
int B::* mp_b = &B::b;
int D::* mp_d = mp_b;       // conversion across a
virtual base
```

825 *declaration cannot be in the same scope as namespace '%S'*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:

```
namespace x {
    int q;
};
int x;
```

826 *'%S' cannot be in the same scope as a namespace*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:

```
int x;
namespace x {
    int q;
};
```

827 *File: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the name of the file in which an error or warning was detected. The message precedes a group of one or more messages written for the file in question. Within each group, references within the file have the format `(line[,column])`.

828 *%s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected either before or after the source file was read during the compilation process.

829 *%s: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while processing the switches specified in a command file or by the contents of an environment variable. The switch that was being processed is displayed following the name of the file or the environment variable.

830 *%s: %S*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while generating a function, such as a constructor, destructor, or assignment operator or while generating the machine instructions for a function which has been analysed. The name of the function is given following text indicating the context from which the message originated.

831 *possible override is '%S'*

The indicated function is ambiguous since that name was defined in more than one base class and one or more of these functions is virtual. Consequently, it cannot be decided which is the virtual function to be used in a class derived from these base classes.

832 *function being overridden is '%S'*

This informational message indicates a function which cannot be overridden by a virtual function which has ellipsis parameters.

833 *name does not reference a namespace*

A ***namespace*** alias definition must reference a ***namespace*** definition.

Example:

```
typedef int T;
namespace a = T;
```

834 *namespace alias cannot be changed*

A ***namespace*** alias definition cannot change which ***namespace*** it is referencing.

Example:

```
namespace ns1 { int x; }
namespace ns2 { int x; }
namespace a = ns1;
namespace a = ns2;
```

835 *cannot throw undefined class object*

C++ does not allow undefined classes to be copied and so an undefined class object cannot be specified in a ***throw*** expression.

836 *symbol has different type than previous symbol in same declaration*

This warning indicates that two symbols in the same declaration have different types. This may be intended but it is often due to a misunderstanding of the C++ declaration syntax.

Example:

```
// change to:
//   char *p;
//   char q;
// or:
//   char *p, *q;
char* p, q;
```

837 *companion definition is '%S'*

This informational message indicates the other symbol that shares a common base type in the same declaration.

838 *syntax error; default argument cannot be processed*

The default argument contains unbalanced braces or parenthesis. The default argument cannot be processed in this form.

839 *default argument started %L*

This informational message indicates where the default argument started so that any problems with missing braces or parenthesis can be fixed quickly and easily.

Example:

```
struct S {
    int f( int t= (4+(3-7), // missing parenthesis
    );
};
```

840 *'%N' cannot be declared in a namespace*

A ***namespace*** cannot contain declarations or definitions of ***operator new*** or ***operator delete*** since they will never be called implicitly in a ***new*** or ***delete*** expression.

Example:

```
namespace N {
    void *operator new( unsigned );
    void operator delete( void * );
};
```

841 *namespace cannot be defined in a non-namespace scope*

A ***namespace*** can only be defined in either the global namespace scope (file scope) or a namespace scope.

Example:

```
struct S {
    namespace N {
    int x;
    };
}
```

842 *namespace '::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** functions. A ***namespace*** qualified name can only be declared in a namespace scope that encloses the qualified name's namespace.

Example:

```
namespace M {
    namespace N {
    void f();
    void g();
    namespace O {
        void N::f() {
          // error
        }
    }
    }
    void N::g() {
      // OK
    }
}
```

843 *cannot cast away volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove volatility from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S volatile * ps;
extern int volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove volatility.

844 *cannot cast away constness and volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness and volatility from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S const volatile * ps;
extern int const volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness and volatility.

845 *cannot cast away unaligned*

A cast or implicit conversion is illegal because a conversion to the target type would add alignment to a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S _unaligned * ps;
extern int _unaligned S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to add alignment.

846 *subscript expression must be integral*

Both of the operands of the indicated index expression are pointers. There may be a missing indirection or function call.

Example:

```
int f();
int *p;
int g() {
    return p[f];
}
```

847 *extension: non-standard user-defined conversion*

An extended conversion was allowed. The latest draft of the C++ working paper does not allow a user-defined conversion to be used in this context. As an extension, the WATCOM compiler supports the conversion since substantial legacy code would not compile without the extension.

848 *useless using directive ignored*

This warning indicates that for most purposes, the ***using namespace*** directive can be removed.

Example:

```
namespace A {
    using namespace A;  // useless
};
```

849 *base class virtual function has not been overridden*

This warning indicates that a virtual function name has been overridden but in an incomplete manner, namely, a virtual function signature has been omitted in the overriding class.

Example:

```
struct B {
    virtual void f() const;
};
struct D : B {
    virtual void f();
};
```

850 *virtual function is '%S'*

This message indicates which virtual function has not been overridden.

851 *macro '%s' defined %L*

This informational message indicates where the macro in question was defined. The message is displayed following an error or warning diagnostic for the macro in question.

Example:

```
#define mac(a,b,c) a+b+c

int i = mac(6,7,8,9,10);
```

The expansion of macro `mac` is erroneous because it contains too many arguments. The informational message will indicate where the macro was defined.

852 *expanding macro '%s' defined %L*

These informational messages indicate the macros that are currently being expanded, along with the location at which they were defined. The message(s) are displayed following a diagnostic which is issued during macro expansion.

853 *conversion to common class type is impossible*

The conversion to a common class is impossible. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type is impossible.

854 *conversion to common class type is ambiguous*

The conversion to a common class is ambiguous. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : public A { B(); };
class C : public A { C(); };
class D : public B, public C { D(); };
extern A a;
extern D d;
int i = ( a == d );
```

The last statement is erroneous since a conversion to a common class type is ambiguous.

855 *conversion to common class type requires private access*

The conversion to a common class violates the access permission which was private. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : private A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (private) access permission.

856 *conversion to common class type requires protected access*

The conversion to a common class violates the access permission which was protected. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : protected A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (protected) access permission.

857 *namespace lookup is ambiguous*

A lookup for a name resulted in two or more non-function names being found. This is not allowed according to the C++ working paper.

Example:

```
namespace M {
    int i;
}
namespace N {
    int i;
    using namespace M;
}
void f() {
    using namespace N;
    i = 7;  // error
}
```

858 *ambiguous namespace symbol is '%S'*

This informational message shows a symbol that conflicted with another symbol during a lookup.

859 *attempt to static_cast from a private base class*

An attempt was made to static_cast a pointer or reference to a private base class to a derived class.

Example:

```
struct PrivateBase {
};

struct Derived : private PrivateBase {
};

extern PrivateBase* pb;
extern PrivateBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a private base class.

860 *attempt to static_cast from a protected base class*

An attempt was made to static_cast a pointer or reference to a protected base class to a derived class.

Example:

```
struct ProtectedBase {
};

struct Derived : protected ProtectedBase {
};

extern ProtectedBase* pb;
extern ProtectedBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a protected base class.

861 *qualified symbol cannot be defined in this scope*

This message indicates that the scope of the symbol is not nested in the current scope. This is a restriction in the C++ language.

Example:

```
namespace A {
    struct S {
    void ok();
    void bad();
    };
    void ok();
    void bad();
};
void A::S::ok() {
}
void A::ok() {
}
namespace B {
    void A::S::bad() {
    // error!
    }
    void A::bad() {
    // error!
    }
};
```

862 *using declaration references non-member*

This message indicates that the entity referenced by the ***using*** declaration is not a class member even though the ***using*** declaration is in class scope.

Example:

```
namespace B {
    int x;
};
struct D {
    using B::x;
};
```

863 *using declaration references class member*

This message indicates that the entity referenced by the ***using*** declaration is a class member even though the ***using*** declaration is not in class scope.

Example:

```
struct B {
    int m;
};
using B::m;
```

864 *invalid suffix for a constant*

An invalid suffix was coded for a constant.

Example:

```
__int64 a[] = {
    0i7, // error
    0i8,
    0i15, // error
    0i16,
    0i31, // error
    0i32,
    0i63, // error
    0i64,
};
```

865 *class in using declaration ('%T') must be a base class*

A ***using*** declaration declared in a class scope can only reference entities in a base class.

Example:

```
struct B {
    int f;
};
struct C {
    int g;
};
struct D : private C {
    B::f;
};
```

866 *name in using declaration is already in scope*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

867 *conflict with a previous using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

868 *conflict with current using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

869 *use of '%N' requires build target to be multi-threaded*

The compiler has detected a use of a run-time function that will create a new thread but the current build target indicates only single-threaded C++ source code is expected. Depending on the user's environment, enabling multi-threaded applications can involve using the "-bm" option or selecting multi-threaded applications through a dialogue.

870 *implementation restriction: cannot use 64-bit value in switch statement*

The use of 64-bit values in switch statements has not been implemented.

871 *implementation restriction: cannot use 64-bit value in case statement*

The use of 64-bit values in case statements has not been implemented.

872 *implementation restriction: cannot use __int64 as bit-field base type*

The use of ***__int64*** for the base type of a bit-field has not been implemented.

873 *based function object cannot be placed in non-code segment "%s".*

Use ***__segname*** with the default code segment "_CODE", or a code segment with the appropriate suffix (indicated by informational message).

Example:

```
int __based(__segname("foo")) f() {return 1;}
```

Example:

```
int __based(__segname("_CODE")) f() {return 1;}
```

874 *Use a segment name ending in "%s", or the default code segment "_CODE".*

This informational message explains how to use ***__segname*** to name a code segment.

875 *RTTI must be enabled to use feature (use 'xr' option)*

RTTI must be enabled by specifying the 'xr' option when the compiler is invoked. The error message indicates that a feature such as ***dynamic_cast,*** or ***typeid*** has been used without enabling RTTI.

876 *'typeid' class type must be defined*

The compile-time type of the expression or type must be completely defined if it is a class type.

Example:

```
struct S;
void foo( S *p ) {
    typeid( *p );
    typeid( S );
}
```

877 *cast involves unrelated member pointers*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1 {
    int poo();
};

typedef int (C1::* C1mp )();

C1mp fmp = (C1mp)&D1::poo;
```

The cast on the last line of the example would be diagnosed.

878 *unexpected type modifier found*

A ***__declspec*** modifier was found that could not be applied to an object or could not be used in this context.

Example:

```
__declspec(thread) struct S {
};
```

879 *invalid bit-field name '%N'*

A bit-field can only have a simple identifier as its name. A qualified name is also not allowed for a bit-field.

Example:

```
struct S {
    int operator + : 1;
};
```

880 *%u padding byte(s) added*

This warning indicates that some extra bytes have been added to a class in order to align member data to its natural alignment.

Example:

```
#pragma pack(push,8)
struct S {
    char c;
    double d;
};
#pragma pack(pop);
```

881 *cannot be called with a '%T *'*

This message indicates that the virtual function cannot be called with a pointer or reference to the current class.

882 *cast involves an undefined member pointer*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host class of at least one member pointer was not a fully defined class and, as such, it is unknown whether the host classes are related through derivation. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1;

typedef int (C1::* C1mp )();
typedef int (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
// D1 may derive from C1
```

The cast on the last line of the example would be diagnosed.

883 *cast changes both member pointer object and class type*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host classes of the member pointers are related through derivation and the object type is also being changed. The cast can be broken up into two casts, one that changes the host class without changing the object type, and another that changes the object type without changing the host class.

Example:

```
struct C1 {
    int fn1();
};
struct D1 : C1 {
    int fn2();
};

typedef int (C1::* C1mp )();
typedef void (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
```

The cast on the last line of the example would be diagnosed.

884 *virtual function '%S' has a different calling convention*

This error indicates that the calling conventions specified in the virtual function prototypes are different. This means that virtual function calls will not function properly since the caller and callee may not agree on how parameters should be passed. Correct the problem by deciding on one calling convention and change the erroneous declaration.

Example:

```
struct B {
    virtual void __cdecl foo( int, int );
};
struct D : B {
    void foo( int, int );
};
```

885 *#endif matches #if in different source file*

This warning may indicate a ***#endif*** nesting problem since the traditional usage of ***#if*** directives is confined to the same source file. This warning may often come before an error and it is hoped will provide information to solve a preprocessing directive problem.

886 *preprocessing directive found %L*

This informational message indicates the location of a preprocessing directive associated with the error or warning message.

887 *unary '-' of unsigned operand produces unsigned result*

When a unary minus ('-') operator is applied to an unsigned operand, the result has an unsigned type rather than a signed type. This warning often occurs because of the misconception that '-' is part of a numeric token rather than as a unary operator. The work-around for the warning is to cast the unary minus operand to the appropriate signed type.

Example:

```
extern void u( int );
extern void u( unsigned );
void fn( unsigned x ) {
    u( -x );
    u( -2147483648 );
}
```

888 *trigraph expansion produced '%c'*

Trigraph expansion occurs at a very low-level so it can affect string literals that contain question marks. This warning can be disabled via the command line or ***#pragma warning*** directive.

Example:

```
// string expands to "(?]?~????"!
char *e = "(???)???-????";
// possible work-arounds
char *f = "(" "???" ")" "???" "-" "????";
char *g = "(\?\?\?)\?\?\?-\?\?\?\?";
```

889 *hexadecimal escape sequence out of range; truncated*

This message indicates that the hexadecimal escape sequence produces an integer that cannot fit into the required character type.

Example:

```
char *p = "\x0aCache Timings\x0a";
```

890 *undefined macro '%s' evaluates to 0*

The ISO C/C++ standard requires that undefined macros evaluate to zero during preprocessor expression evaluation. This default behaviour can often mask incorrectly spelled macro references. The warning is useful when used in critical environments where all macros will be defined.

Example:

```
#if _PRODUCTI0N // should be _PRODUCTION
#endif
```

891 *char constant has value %u (more than 8 bits)*

The ISO C/C++ standard requires that multi-char character constants be accepted with an implementation defined value. This default behaviour can often mask incorrectly specified character constants.

Example:

```
int x = '\0x1a'; // warning
int y = '\x1a';
```

892 *promotion of unadorned char type to int*

This message is enabled by the hidden -jw option. The warning may be used to locate all places where an unadorned char type (i.e., a type that is specified as ***char*** and neither ***signed char*** nor ***unsigned char***). This may cause portability problems since compilers have freedom to specify whether the unadorned char type is to be signed or unsigned. The promotion to ***int*** will have different values, depending on the choice being made.

893 *switch statement has no case labels*

The switch statement referenced in the warning did not have any case labels. Without case labels, a switch statement will always jump to the default case code.

Example:

```
void fn( int x )
{
    switch( x ) {
    default:
    ++x;
    }
}
```

894 *unexpected character (%u) in source file*

The compiler has encountered a character in the source file that is not in the allowable set of input characters. The decimal representation of the character byte is output for diagnostic purposes.

Example:

```
// invalid char '\0'
```

895 *ignoring whitespace after line splice*

The compiler is ignoring some whitespace characters that occur after the line splice. This warning is useful when the source code must be compiled with other compilers that do not allow this extension.

Example:

```
#define XXXX int  \
x;

XXXX
```

896 *empty member declaration*

The compiler is warning about an extra semicolon found in a class definition. The extra semicolon is valid C++ but some C++ compilers do not accept this as valid syntax.

Example:

```
struct S { ; };
```

897 *'%S' makes use of a non-portable feature (zero-sized array)*

The compiler is warning about the use of a non-portable feature in a declaration or definition. This warning is available for environments where diagnosing the use of non-portable features is useful in improving the portability of the code.

Example:

```
struct D {
    int d;
    char a[];
};
```

898 *in-class initialization is only allowed for const static integral members*

Example:

```
struct A {
    static int i = 0;
};
```

899 *cannot convert expression to target type*

The implicit cast is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended cast.

Example:

```
struct T {
};

void fn()
{
    bool b = T;
}
```

900 *unknown template specialization of '%S'*

Example:

```
template<class T>
struct A { };

template<class T>
void A<T *>::f() {
}
```

901 *wrong number of template arguments for '%S'*

Example:

```
template<class T>
struct A { };

template<class T, class U>
struct A<T, U> { };
}
```

902 *cannot explicitly specialize member of '%S'*

Example:

```
template<class T>
struct A { };

template<>
struct A<int> {
    void f();
};

template<>
void A<int>::f() {
}
```

903 *specialization arguments for '%S' match primary template*

Example:

```
template<class T>
struct A { };

template<class T>
struct A<T> { };
```

904 *partial template specialization for '%S' ambiguous*

Example:

```
template<class T, class U>
struct A { };

template<class T, class U>
struct A<T *, U> { };

template<class T, class U>
struct A<T, U *> { };

A<int *, int *> a;
```

905 *static assertion failed '%s'*

Example:
```
static_assert( false, "false" );
```

906 *Exported templates are not supported by Open Watcom C++*

Example:
```
export template< class T >
struct A {
};
```

907 *redeclaration of member function '%S' not allowed*

Example:
```
struct A {
    void f();
    void f();
};
```

908 *candidate defined %L*

909 *Invalid register name '%s' in #pragma*

The register name is invalid/unknown.

910 *Archaic syntax: class/struct missing in explicit template instantiation*

Archaic syntax has been used. The standard requires a ***class*** or ***struct*** keyword to be used.

Example:
```
template< class T >
class MyTemplate { };

template MyTemplate< int >;
```

Example:
```
template class MyTemplate< int >;
```

911 *destructor for type void cannot be called*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot destruct an instance of ***void.***

912 *'typename' keyword used outside template*

The ***typename*** keyword is only allowed inside templates.

913 *'%N' does not have a return type specified*

In C++, functions must have an explicit return type specified, default int type is no longer assumed.

Example:
```
f ();
```

914 *'main' must return 'int'*

The "main" function shall have a return type of type int.

Example:
```
void main()
{ }
```

915 *explicit may only be used within class definition*

The explicit specifier shall be used only in the declaration of a constructor within its class definition.

Example:
```
struct A {
    explicit A();
};

explicit A::A()
{ }
```

916 *virtual may only be used within class definition*

The virtual specifier shall be used only in the initial declaration of a class member function.

Example:
```
struct A {
    virtual void f();
};

virtual void A::f()
{ }
```

917 *cannot redefine default template argument '%N'*

A template-parameter shall not be given default arguments by two different declarations in the same scope.

Example:
```
template< class T = int >
class X;

template< class T = int >
class X {
};
```

918 *cannot have default template arguments in partial specializations*

A partial specialization cannot have default template arguments.

Example:

```
template< class T >
class X {
};

template< class T = int >
class X< T * > {
};
```

919 *delete of a pointer to void*

If the dynamic type of the object to be deleted differs from its static type, the behavior is undefined. This implies that an object cannot be deleted using a pointer of type void* because there are no objects of type void.

Example:

```
void fn( void *p, void *q ) {
    delete p;
    delete [] q;
}
```

920 *'long char' is deprecated, use wchar_t instead*

The standard C++ 'wchar_t' type specifier should be used instead of the Open Watcom specific 'long char' type specifier.

Example:

```
void fn( ) {
    long char c;
}
```

921 *namespace '%I' not allowed in using-declaration*

Specifying a namespace-name is not allowed in a using-declaration, a using-directive must be used instead.

Example:

```
namespace ns { }
using ns;
```

922 *candidate %C defined %L*

923 *qualified name '%I' does not name a class*

Example:

```
namespace ns {
}
struct ns::A {
};
```

924 *expected class type, but got '%T'*

Example:

```
template< class T >
struct A : public T {
};

A< int > a;
```

925 *syntax error near '%s'; probable cause: incorrectly spelled type name*

The identifier in the error message has not been declared as a type name in any scope at this point in the code. This may be the cause of the syntax error.

926 *syntax error: '%s' has not been declared as a member*

The identifier in the error message has not been declared as member. This may be the cause of the syntax error.

Example:

```
struct A { };

void fn() {
    A::undeclared = 0;
}
```

927 *syntax error: '%s' has not been declared*

The identifier in the error message has not been declared. This may be the cause of the syntax error.

Example:

```
void fn() {
    ::undeclared = 0;
}
```

928 *syntax error: identifier '%s', but expected: '%s'*

929 *syntax error: token '%s', but expected: '%s'*

930 *member '%S' cannot be declared in this class*

A member cannot be declared with the same name as its containing class if the class has a user-declared constructor.

Example:

```
struct S {
    S() { }
    int S;   // Error!
};
```

931 *cv-qualifier in cast to '%T' is meaningless*

A top-level cv-qualifier for a non-class rvalue is meaningless.

Example:

```
const int i = (const int) 0;
```

932 *cv-qualifier in return type '%T' is meaningless*

A top-level cv-qualifier for a non-class rvalue is meaningless.

Example:

```
const int f() {
    return 0;
}
```

933 *use of C-style cast to '%T' is discouraged*

Use of C-style casts "(type) (expr)" is discouraged in favour of explicit C++ casts like static_cast, const_cast, dynamic_cast and reinterpret_cast.

Example:

```
const signed int *f( unsigned int *psi ) {
    return ( signed int * ) psi;
}
```

934 *unable to match function template definition '%S'*

The function template definition cannot be matched to an earlier declaration.

Example:

```
template< class T >
struct A
{
    A( );
};

template< class T >
A< int >::A( )
{ }
```

935 *form is '#pragma enable_message(msgnum)'*

This ***pragma*** enables the specified warning message.

936 *form is '#pragma disable_message(msgnum)'*

This ***pragma*** disables the specified warning message.

937 *option requires a character*

The specified option is not recognized by the compiler since there was no character after it (i.e., "-p#@").

938 *'auto' is no longer a storage specifier in C++11 mode*

When C++11 is enabled, the ***auto*** can no longer appear as a storage specifier.

939 *Implicit conversion from 'decltype(nullptr)' to 'bool'.*

When C++11 is enabled, an implicit conversion from std::nullptr_t to bool is suspicious.