

```
{  
    FILE* sfile;  
    int count = 0;  
  
    sfile = fopen("file1",  
    "r");  
  
    if( sfile == NULL)  
    {  
        return -1;  
    }  
  
    while (1)  
    {  
        char c;  
        c = fgetc(  
        if(c == EOF)  
        {  
            break;  
        }  
        else  
        {  
            count++;  
        }  
    }  
  
    return count;  
}
```

C166/ST10 v8.5

C++ Compiler User's Manual

A publication of
Altium BV
Documentation Department
Copyright © 2004 Altium BV

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Macrovision Corporation.
HP and HP-UX are trademarks of Hewlett-Packard Co.
IBM is a trademark of International Business Machines Corp.
Intel is a trademark of Intel Corporation.
Motorola is a trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

The STLport C++ library has the following copyrights:

Copyright © 1994 Hewlett-Packard Company
Copyright © 1996,97 Silicon Graphics Computer Systems, Inc.
Copyright © 1997 Moscow Center for SPARC Technology
Copyright © 1999, 2000 Boris Fomitchev

Data subject to alteration without notice.

<http://www.tasking.com>
<http://www.altium.com>

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

Altium reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS



TASKING



CONTENTS

OVERVIEW 1-1

1.1	Introduction to C++ Compiler	1-3
1.2	Development Structure	1-4
1.2.1	The Prelinker Phase	1-5
1.2.2	The Muncher Phase	1-7
1.3	Environment Variables	1-8
1.4	File Extensions	1-9

LANGUAGE IMPLEMENTATION 2-1

2.1	Introduction	2-3
2.2	C++ Library	2-3
2.3	C++ Language Extension Keywords	2-4
2.4	C++ Dialect Accepted	2-7
2.4.1	New Language Features Accepted	2-7
2.4.2	New Language Features Not Accepted	2-10
2.4.3	Anachronisms Accepted	2-10
2.4.4	Extensions Accepted in Normal C++ Mode	2-12
2.4.5	Extensions Accepted in Cfront 2.1 Compatibility Mode	2-14
2.4.6	Extensions Accepted in Cfront 2.1 and 3.0 Compatibility Mode	2-18
2.5	Namespace Support	2-24
2.6	Template Instantiation	2-26
2.6.1	Automatic Instantiation	2-27
2.6.2	Instantiation Modes	2-31
2.6.3	Instantiation #pragma Directives	2-32
2.6.4	Implicit Inclusion	2-35
2.7	Predefined Macros	2-36
2.8	Precompiled Headers	2-38
2.8.1	Automatic Precompiled Header Processing	2-38
2.8.2	Manual Precompiled Header Processing	2-42
2.8.3	Other Ways to Control Precompiled Headers	2-43
2.8.4	Performance Issues	2-44
2.9	Prohibited c166 Optimizations	2-46
2.9.1	'main' Labels in a C++ Application	2-46

2.9.2 Prohibited c166 Optimizations 2-46

COMPILER USE 3-1

3.1 Invocation 3-3

3.1.1 Detailed Description of the Compiler Options 3-16

3.2 Include Files 3-113

3.3 Pragmas 3-116

3.4 Compiler Limits 3-118

COMPILER DIAGNOSTICS 4-1

4.1 Diagnostic Messages 4-3

4.2 Termination Messages 4-5

4.3 Response to Signals 4-6

4.4 Return Values 4-6

ERROR MESSAGES A-1

1 Introduction A-3

2 Messages A-4

UTILITY PROGRAMS B-1

1 Introduction B-3

2 Prelinker B-3

3 Muncher B-5

INDEX

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING C166/ST10 C++ Compiler. It assumes that you are conversant with the C and C++ language.

MANUAL STRUCTURE

Related Publications
Conventions Used In This Manual

Chapters

1. Overview
Provides an overview of the TASKING C166/ST10 toolchain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build an application from your C++ file.
2. Language Implementation
Concentrates on the approach of the C166/ST10 architecture and describes the language implementation. The C++ language itself is not described in this document.
3. Compiler Use
Deals with invocation, command line options and pragmas.
4. Compiler Diagnostics
Describes the exit status and error/warning messages of the C++ compiler.

Appendices

- A. Error Messages
Contains an overview of the error messages.
- B. Utility Programs
Contains a description of the prelinker and the muncher which are delivered with the C++ compiler package.

RELATED PUBLICATIONS

- The C++ Programming Language (second edition)
by Bjarne Strastrup (1991, Addison Wesley)
- ISO/IEC 14882:1998 C++ standard [ANSI]
More information on the standards can be found at
<http://www.ansi.org>
- The Annotated C++ Reference Manual
by Margaret A. Ellis and Bjarne Strastrup (1990, Addison Wesley)
- The C Programming Language (second edition)
by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159–1989 standard [ANSI]
- C166/ST10 C Cross–Compiler User’s Manual [TASKING,
MA019–002–00–00]
- C166/ST10 Cross–Assembler, Linker/Locator, Utilities User’s Manual
[TASKING, MA019–000–00–00]
- C166/ST10 CrossView Pro Debugger User’s Manual [TASKING,
MA019–041–00–00]

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{ } Items shown inside curly braces enclose a list from which you must choose an item.

[] Items shown inside square brackets enclose items that are optional.

| The vertical bar separates items in a list. It can be read as OR.

italics Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

filename

means: type the name of your file in place of the word *filename*.

... An ellipsis indicates that you can repeat the preceding item zero or more times.

screen font Represents input examples and screen output examples.

bold font Represents a command name, an option or a complete command line which you can enter.

For example

command [*option*]... *filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.

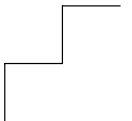
CHAPTER

1

OVERVIEW



TASKING



1

CHAPTER

1.1 INTRODUCTION TO C++ COMPILER

This manual provides a functional description of the TASKING C166/ST10 C++ Compiler. This manual uses **cp166** (the name of the binary) as a shorthand notation for "TASKING C166/ST10 C++ Compiler". You should be familiar with the C++ language and with the ANSI/ISO C language.

The C++ compiler can be seen as a preprocessor or front end which accepts C++ source files or sources using C++ language features. The output generated by **cp166** is C166/ST10 C, which can be translated with the C compiler **c166**.

The C++ compiler is part of a complete toolchain. For details about the C compiler see the "C Compiler User's Manual".

The C++ compiler is normally invoked via the control program which is part of the toolchain. The control program facilitates the invocation of various components of the toolchain. The control program recognizes several filename extensions. C++ source files (**.cc**, **.cxx**, **.cpp** or **.c** with the **-c++** option) are passed to the C++ compiler. C source files (**.c**) are passed to the compiler. Assembly source files (**.asm**) are preprocessed and passed to the assembler. Assembly sources (**.src**) are directly passed to the assembler. Relocatable object files (**.obj**) and libraries (**.lib**) are recognized as linker input files. Files with extension **.lno** and **.ilo** are treated as locator input files. The control program supports options to stop at any stage in the compilation process and has options to produce and retain intermediate files.

The C++ compiler accepts the C++ language of the ISO/IEC 14882:1998 C++ standard, with some minor exceptions documented in the next chapter. With the proper command line options, it alternatively accepts the ANSI/ISO C language or traditional K&R C (B. W. Kernighan and D. M. Ritchie). It also accepts embedded C++ language extensions.

The C++ compiler does no optimization. Its goal is to produce quickly a complete and clean parsed form of the source program, and to diagnose errors. It does complete error checking, produces clear error messages (including the position of the error within the source line), and avoids cascading of errors. It also tries to avoid seeming overly finicky to a knowledgeable C or C++ programmer.

1.2 DEVELOPMENT STRUCTURE

The next figure explains the relationship between the different parts of the C166/ST10 toolchain:

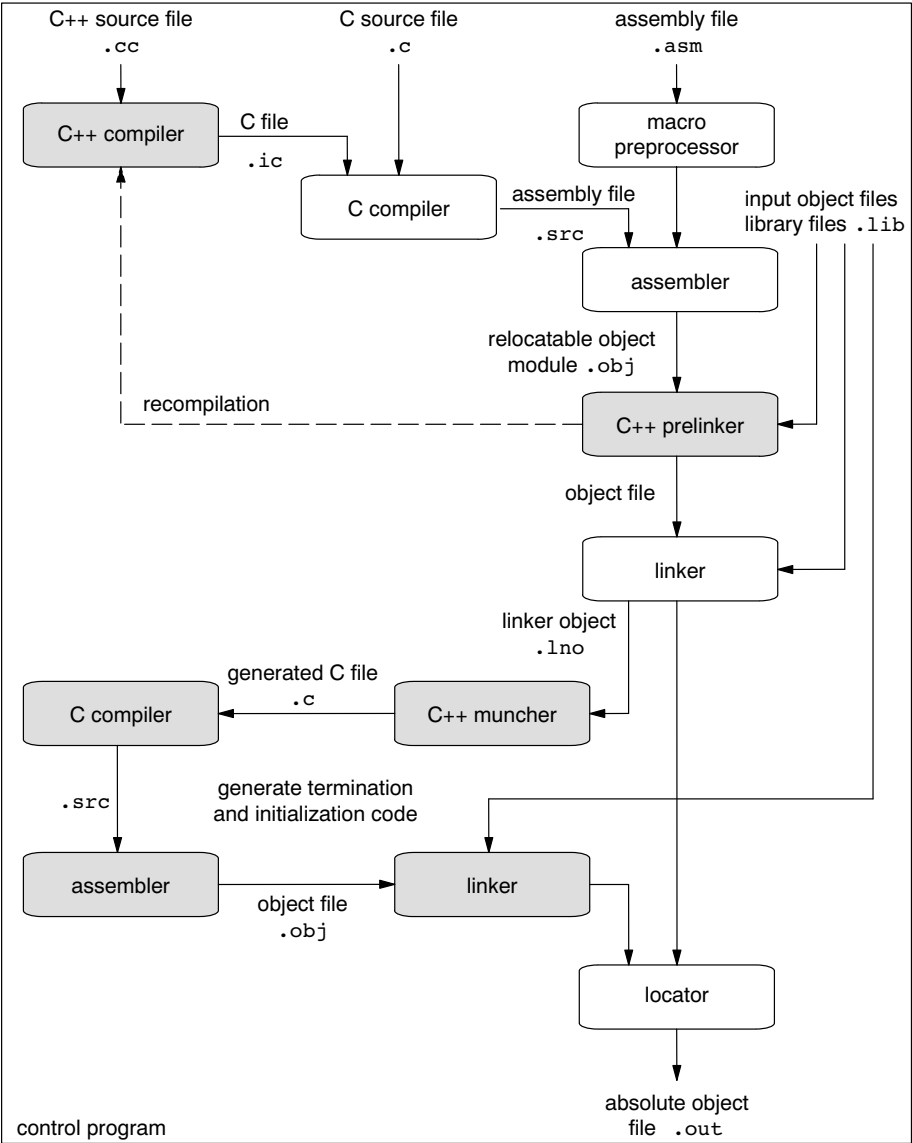


Figure 1-1: Development flow

1.2.1 THE PRELINKER PHASE

The C++ compiler provides a complete prototype implementation of an automatic instantiation mechanism. The automatic instantiation mechanism is a "linker feedback" mechanism. It works by providing additional information in the object file that is used by a "prelinker" to determine which template entities require instantiation so that the program can be linked successfully. Unlike most aspects of the C++ compiler the automatic instantiation mechanism is, by its nature, dependent on certain operating system and object file format properties. In particular, the prelinker is a separate program that accesses information about the symbols defined in object files.

At the end of each compilation, the C++ compiler determines whether any template entities were referenced in the translation unit. If so, an "instantiation information" file is created, referred to for convenience as a `.ii` file. If no template entities were referenced in the translation unit, the `.ii` file will not be created and any existing file will be removed. If an error occurs during compilation, the state of the `.ii` file is unchanged.

Once a complete set of object files has been generated, including the appropriate flags, the prelinker is invoked to determine whether any new instantiations are required or if any existing instantiations are no longer required. The command line arguments to the prelinker include a list of input files to be analyzed. The input files are the object files and libraries that constitute the application. The prelinker begins by looking for instantiation information files for each of the object files. If no instantiation information files are present, the prelinker concludes that no further action is required.

If there are instantiation information files, the prelinker reads the current instantiation list from each information file. The instantiation list contains the list of instantiations assigned to a given source file by a previous invocation of the prelinker. The prelinker produces a list of the global symbols that are referenced or defined by each of the input files. The prelinker then simulates a link operation to determine which symbols must be defined for the application to link successfully.

When the link simulation has been completed, the prelinker processes each input file to determine whether any new instantiations should be assigned to the input file or if any existing instantiations should be removed. The prelinker goes through the current instantiation list from the instantiation information file to determine whether any of the existing instantiations are no longer needed. An instantiation may be no longer needed because the template entity is no longer referenced by the program or because a user supplied specialization has been provided. If the instantiation is no longer needed, it is removed from the list (internally; the file will be updated later) and the file is flagged as requiring recompilation.

The prelinker then examines any symbols referenced by the input file. The responsibility for generating an instantiation of a given entity that has not already been defined is assigned to the first file that is capable of generating that instantiation.

Once all of the assignments have been updated, the prelinker once again goes through the list of object files. For each, if the corresponding instantiation information file must be updated, the new file is written. Only source files whose corresponding `.ii` file has been modified will be recompiled.

At this point each `.ii` file contains the information needed to recompile the source file and a list of instantiations assigned to the source file, in the form of mangled function and static data member names.

If an error occurs during a recompilation, the prelinker exits without updating the remaining information files and without attempting any additional compilations.

If all recompilations complete without error, the prelink process is repeated, since an instantiation can produce the demand for another instantiation. This prelink cycle (finding uninstantiated templates, updating the appropriate `.ii` files, and dispatching recompilations) continues until no further recompilations are required.

When the prelinker is finished, the linker is invoked. Note that simply because the prelinker completes successfully does not assure that the linker will not detect errors. Unresolvable template references and other linker errors will not be diagnosed by the prelinker.

1.2.2 THE MUNCHER PHASE

The C++ muncher implements global initialization and termination code.

The muncher accepts the output of the linker as its input file. It generates a C program that defines a data structure containing a list of pointers to the initialization and termination routines. This generated program is then compiled and linked in with the executable. The data structure is consulted at run-time by startup code invoked from `_main`, and the routines on the list are invoked at the appropriate times.

1.3 ENVIRONMENT VARIABLES

This section contains an overview of the environment variables used by the C166/ST10 toolchain.

Environment Variable	Description
A166INC	Specifies an alternative path for STDNAMES files for the assembler a166 .
C166INC	Specifies an alternative path for #include files for the C compiler c166 .
CC166BIN	When this variable is set, the control program cc166 , prepends the directory specified by this variable to the names of the tools invoked.
CC166OPT	Specifies extra options and/or arguments to each invocation of cc166 . The control program processes the arguments from this variable before the command line arguments.
CP166INC	Specifies an alternative path for #include files for the C++ compiler cp166 .
LINK166	Specifies extra options and/or arguments to each invocation of the link stage of l166 .
LOCATE166	Specifies extra options and/or arguments to each invocation of the locate stage of l166 .
M166INC	Specifies an alternative path for include files for the macro preprocessor m166 .
LM_LICENSE_FILE	With this variable you specify the location of the license data file. You only need to specify this variable if your host uses the FLEXlm licence manager.
TASKING_LIC_WAIT	If you set this variable, the tool will wait for a license to become available, if all licenses are taken. If you have not set this variable, the tool aborts with an error message.
PATH	With this variable you specify the directory in which the executables reside (default: <i>product\bin</i>). This allows you to call the executables when you are not in the <i>bin</i> directory.
TMPDIR	With this variable you specify the location where programs can create temporary files.

Table 1-1: Environment variables

1.4 FILE EXTENSIONS

For compatibility with future TASKING Cross-Software the following extensions are suggested:

Source files:

.cc	C++ source file, input for C++ compiler
.cxx	C++ source file, input for C++ compiler
.cpp	C++ source file, input for C++ compiler
.c	C source file, input for C compiler (or for C++ compiler if you use the -c++ option of the control program)
.asm	hand-written assembly source file, input for the assembler

Generated source files:

.ic	temporary C source file generated by the C++ compiler, input for the C compiler
.src	assembly source file generated by the C compiler, input for the assembler

Object files:

.obj	relocatable IEEE-695 object file generated by the assembler, input for the linker
.lno	linked object module
.lib	object library file
.out	absolute locator output file
.abs	absolute IEEE-695 object file
.hex	absolute Intel Hex object file

List files:

.mpl	macro preprocessor list file
.mpe	macro preprocessor error list file
.lst	assembler list file
.lnl	linker map file
.map	locator map file

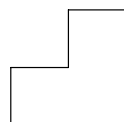
CHAPTER

2

LANGUAGE IMPLEMENTATION



TASKING



2

CHAPTER

2.1 INTRODUCTION

The TASKING C++ compiler (**cp166**) offers a new approach to high-level language programming for the C166/ST10 family. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:1998 standard, with the exceptions listed in section 2.4. It also accepts the language extensions of the C compiler.

This chapter describes the C++ language extensions and some specific features.

2.2 C++ LIBRARY

The TASKING C++ compiler supports the STLport C++ libraries. STLport is a multiplatform ANSI C++ Standard Library implementation. It is a free, open-source product, which is delivered with the TASKING C++ compiler. The library supports standard templates and I/O streams.

The include files for the STLport C++ libraries are present in directory **include.stl** relative to the product installation directory.

You can find more information and documentation on the STLport library on the following site:

<http://www.stlport.org/doc/index.html>

Also read the license agreement on:

<http://www.stlport.org/doc/license.html>

This license agreement is applicable to the C++ library only. All other product components fall under the TASKING license agreement.

For an STL Programmer's Guide you can see:

<http://www.sgi.com/tech/stl/index.html>



The following C++ libraries are delivered with the product:

Library to link	Description
cp166t.lib cp166s.lib cp166m.lib cp166l.lib cp166h.lib	C++ library for each memory model (tiny, small, medium, large, huge)
cp166tx.lib cp166sx.lib cp166mx.lib cp166lx.lib cp166hx.lib	C++ library with exception handling
stl166s.lib stl166l.lib stl166h.lib	STLport library (small, large or huge)
stl166sx.lib stl166lx.lib stl166hx.lib	STLport library with exception handling
stlo166s.lib stlo166l.lib stlo166h.lib	Optional part of STLport library (small, large or huge). I/O streams, monetary, locale and number punctuation support.
stlo166sx.lib stlo166lx.lib stlo166hx.lib	Optional part of STLport library with exception handling

2.3 C++ LANGUAGE EXTENSION KEYWORDS

The C++ compiler supports the same language extension keywords as the C compiler. These language extensions are enabled by default (**--embedded**), but you can disable them by specifying the **--no-embedded** command line option. When **-A** is used, the extensions will be disabled.

The following language extensions are supported:

bit

You can use data type `_bit` for the type definition of scalars and for the return type of functions.

_bitword

You can declare word variables in the bit-addressable area as fp. You can access individual bits using the intrinsic functions `_getbit()` and `_putbit()`.

_sfrbit / _esfrbit

Data types for the declaration of specific, absolute bits in special function registers or special absolute bits in the SFR address space.

_sfr / _esfr

Data types for the declaration of Special Function Registers.

_xsfr

Data types for the declaration of Special Function Registers not residing in SFR memory but do reside in internal RAM. An example of these SFRs are PEC source and destination pointers. The compiler will use a 'mem' addressing mode for this data type whereas for an object of type `_sfr` a 'reg' or 'mem' addressing mode may be used.

These SFRs are not bitaddressable.

_at

You can specify a variable to be at an absolute address.

_atbit

You can specify a variable to be at a bit offset within a `_bitword` or bitaddressable `_sfr` variable.

_inline

Used for defining inline functions.

_usm / _nousm

With these function qualifiers you can force that a function is called using the user stack model calling convention or using the generic CALL/RET calling convention.

_bita

You can tell the C++ compiler that a struct must be located in bitaddressable memory by using the `_bita` memory qualifier.

memory-specific pointers

cp166 allows you to define pointers which point to a specific target memory. These types of pointers are very efficient and require only 2 or 4 bytes memory space.

special types

Apart from a memory category (extern, static, ...) you can specify a storage type in each declaration. This way you obtain a memory model-independent addressing of variables in several address ranges of the C166/ST10 (**_near**, **_xnear**, **_far**, **_huge**, **_shuge**, **_system**, **_iram**).

interrupt functions

You can specify interrupt functions directly through interrupt vectors in the C++ language (**_interrupt** keyword). You may also specify the register bank to be used (**_using** keyword).

intrinsic functions

A number of pre-declared functions can be used to generate inline assembly code at the location of the intrinsic (built-in) function call. This avoids the overhead which is normally used to do parameter passing and context saving before executing the called function.

pragmas

The C++ compiler supports the same pragmas as the C compiler. Pragmas give directions to the code generator of the compiler.

All of the language extensions mentioned above are described in detail in the *C Cross-Compiler User's Manual*.

2.4 C++ DIALECT ACCEPTED

The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:1998 standard, with the exceptions listed below.

The C++ compiler also has a cfront compatibility mode, which duplicates a number of features and bugs of cfront 2.1 and 3.0.x. Complete compatibility is not guaranteed or intended; the mode is there to allow programmers who have unwittingly used cfront features to continue to compile their existing code. In particular, if a program gets an error when compiled by cfront, the C++ compiler may produce a different error or no error at all.

Command line options are also available to enable and disable anachronisms and strict standard-conformance checking.

2.4.1 NEW LANGUAGE FEATURES ACCEPTED

The following features not in traditional C++ (the C++ language of *The Annotated C++ Reference Manual* by Ellis and Stroustrup (ARM)) but in the standard are implemented:

- The dependent statement of an **if**, **while**, **do-while**, or **for** is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an **if**, **while**, **do-while**, or **for**, as the first operand of a **"?"** operator, or as an operand of the **"&&"**, **":"**, or **"!"** operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form **x.::A::B** and **p->::A::B**.
- The precedence of the third operand of the **"?"** operator is changed.
- If control reaches the end of the **main()** routine, and **main()** has an integral return type, it is treated as if a **return 0;** statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.

- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions, such as conversion from `T**` to `T const * const *` are allowed.
- Digraphs are recognized.
- Operator keywords (e.g., `not`, `and`, `bitand`, etc.) are recognized.
- Static data member declarations can be used to declare member constants.
- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (run-time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on non-static data member declarations.
- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare non-converting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.

- The new specialization syntax (using “**template** <>”) is implemented.
- Cv-qualifiers are retained on rvalues (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- **extern inline** functions are supported, and the default linkage for **inline** functions is external.
- A typedef name may be used in an explicit destructor call.
- Placement delete is implemented.
- An array allocated via a placement new can be deallocated via delete.
- Covariant return types on overriding virtual functions are supported.
- **enum** types are considered to be non-integral types.
- Partial specialization of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.
- It is possible to overload operators using functions that take **enum** types and no **class** types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form **x.A::B** and **p->A::B** are supported.
- The notation **:: template** (and **->template**, etc.) is supported.
- In a reference of the form **f()->g()**, with **g** a static member function, **f()** is evaluated. The ARM specifies that the left operand is not evaluated in such cases.
- **enum** types can contain values larger than can be contained in an **int**.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have **const** type.

- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- Function-try-blocks, i.e., try-blocks that are the top-level statements of functions, constructors, or destructors, are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.
- Template template parameters are implemented.

2.4.2 NEW LANGUAGE FEATURES NOT ACCEPTED

The following features of the C++ standard are not implemented yet:

- Two-phase name binding in templates, as described in [temp.res] and [temp.dep] of the standard, is not implemented.
- The `export` keyword for templates is not implemented.
- A partial specialization of a class member template cannot be added outside of the class definition.

2.4.3 ANACHRONISMS ACCEPTED

The following anachronisms are accepted when anachronisms are enabled (with `--anachronisms`):

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.

- The number of elements in an array may be specified in an array `delete` operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to `this` in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the "assignment to `this`" configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

```
int f(int);  
int f(x) char x; { return x; }
```

Note that in C this code is legal but has a different meaning: a tentative declaration of `f` is followed by its definition.

- When **--nonconst-ref-anachronism** is enabled, a reference to a non-const class can be bound to a class rvalue of the same type or a derived type thereof.

```
struct A {
    A(int);
    A operator=(A&);
    A operator+(const A&);
};
main () {
    A b(1);
    b = A(1) + A(2);  // Allowed as anachronism
}
```

2.4.4 EXTENSIONS ACCEPTED IN NORMAL C++ MODE

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A **friend** declaration for a class may omit the **class** keyword:

```
class A {
    friend B;  // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f();  // Should be int f();
};
```

- The preprocessing symbol **cplusplus** is defined in addition to the standard **__cplusplus**.
- A pointer to a constant type can be **deleted**.

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a default assignment operator, that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is cfront behavior that is known to be relied upon in at least one widely used library.) Here is an example:

```
struct A { };
struct B : public A {
    B& operator=(A&);
};
```

By default, as well as in cfront-compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode `B::operator=(A&)` is not a copy assignment operator and `B::operator=(const B&)` is implicitly declared.

- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. Here's an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)()        // pf points to an extern "C++" function
    = &f;           // error unless implicit conversion is
                    // allowed
```

This extension is allowed in environments where C and C++ functions share the same calling conventions. It is enabled by default; it can also be enabled in cfront-compatibility mode or with option **--implicit-extern-c-type-conversion**. It is disabled in strict-ANSI mode.

- A `"?"` operator whose second and third operands are string literals or wide string literals can be implicitly converted to `"char *"` or `"wchar_t *"`. (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `"char *"`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `"?"` operation is an extension.)

```
char *p = x ? "abc" : "def";
```

- Except in strict-ANSI mode, default arguments may be specified for function parameters other than those of a top-level function declaration (e.g., they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).

2.4.5 EXTENSIONS ACCEPTED IN CFRONT 2.1 COMPATIBILITY MODE

The following extensions are accepted in cfront 2.1 compatibility mode in addition to the extensions listed in the 2.1/3.0 section following (i.e., these are things that were corrected in the 3.0 release of cfront):

- The dependent statement of an **if**, **while**, **do-while**, or **for** is not considered to define a scope. The dependent statement may not be a declaration. Any objects constructed within the dependent statement are destroyed at exit from the dependent statement.
- Implicit conversion from integral types to enumeration types is allowed.
- A non-**const** member function may be called for a **const** object. A warning is issued.
- A **const void *** value may be implicitly converted to a **void *** value, e.g., when passed as an argument.
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion.
- When a built-in operator is considered alongside overloaded operators in overload resolution, the match of an operand of a built-in type against the built-in type required by the built-in operator is considered a standard conversion in all cases (e.g., even when the type is exactly right without conversion).
- A reference to a non-**const** type may be initialized from a value that is a **const**-qualified version of the same type, but only if the value is the result of selecting a member from a **const** class object or a pointer to such an object.
- The cfront 2.1 "transitional model" for nested type support is simulated. In the transitional model a nested type is promoted to the file scope unless a type of the same name already exists at the file scope. It is an error to have two nested classes of the same name that need to be promoted to file scope or to define a type at file scope after the declaration of a nested class of the same name. This "feature" actually restricts the source language accepted by the compiler. This is necessary because of the effect this feature has on the name mangling of functions that use nested types in their signature. This feature does not apply to template classes.

- A cast to an array type is allowed; it is treated like a cast to a pointer to the array element type. A warning is issued.
- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.
- An expression of type `void` may be supplied on the return statement in a function with a void return type. A warning is issued.
- Cfront has a bug that causes a global identifier to be found when a member of a class or one of its base classes should actually be found. This bug is emulated in cfront compatibility mode. A warning is issued when, because of this feature, a nonstandard lookup is performed. The following conditions must be satisfied for the nonstandard lookup to be performed:
 - A member in a base class must have the same name as an identifier at the global scope. The member may be a function, static data member, or non-static data member. Member type names do not apply because a nested type will be promoted to the global scope by cfront which disallows a later declaration of a type with the same name at the global scope.
 - The declaration of the global scope name must occur between the declaration of the derived class and the declaration of an out-of-line constructor or destructor. The global scope name must be a type name.
 - No other member function definition, even one for an unrelated class, may appear between the destructor and the offending reference. This has the effect that the nonstandard lookup applies to only one class at any given point in time. For example:

```
struct B {  
    void func(const char*);  
};
```

```

struct D : public B {
public:
    D();
    void Init(const char* );
};

struct func {
    func( const char* msg);
};

D::D()

void D::Init(const char* t)
{
    //Should call B::func -- calls func::func instead.
    new func(t);
}

```

The global scope name must be present in a base class (**B::func** in this example) for the nonstandard lookup to occur. Even if the derived class were to have a member named **func**, it is still the presence of **B::func** that determines how the lookup will be performed.

- A parameter of type "**const void ***" is allowed on operator **delete**; it is treated as equivalent to "**void ***".
- A period ("**.**") may be used for qualification where "**::**" should be used. Only "**::**" may be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say **A::B::C** or **A.B.C** but not **A::B.C** or **A.B::C**). A period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as **A<T>::B**.
- Cfront 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example function **f** should refer to the functions and variables (e.g., **f1** and **a1**) from the class declaration. Instead, the global definitions are used.

```

int a1;
int e1;
void f1();
class A {
    int a1;
    void f1();
    friend void f()
    {
        int i1 = a1; // cfront uses global a1
        f1(); // cfront uses global f1
    }
};

```

Only the innermost class scope is (incorrectly) skipped by cfront as illustrated in the following example.

```

int a1;
int b1;
struct A {
    static int a1;
    class B {
        static int b1;
        friend void f()
        {
            int i1 = a1; // cfront uses A::a1
            int j1 = b1; // cfront uses global b1
        }
    };
};

```

- `operator=` may be declared as a nonmember function. (This is flagged as an anachronism by cfront 2.1)
- A type qualifier is allowed (but ignored) on the declaration of a constructor or destructor. For example:

```

class A {
    A() const; // No error in cfront 2.1 mode
};

```

2.4.6 EXTENSIONS ACCEPTED IN CFRONT 2.1 AND 3.0 COMPATIBILITY MODE

The following extensions are accepted in both cfront 2.1 and cfront 3.0 compatibility mode (i.e., these are features or problems that exist in both cfront 2.1 and 3.0):

- Type qualifiers on the **this** parameter may to be dropped in contexts such as this example:

```
struct A {
    void f() const;
};
void (A::*fp)() = &A::f;
```

This is actually a safe operation. A pointer to a **const** function may be put into a pointer to non-**const**, because a call using the pointer is permitted to modify the object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.

- Conversion operators specifying conversion to **void** are allowed.
- A nonstandard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in cfront mode the declaration is also allowed to introduce a new type name.

```
struct A {
    friend B;
};
```

- The third operand of the **?** operator is a conditional expression instead of an assignment expression as it is in the modern language.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example,

```
int *p;
const int *&r = p; // No temporary used
```

- A reference may be initialized with a null.
- Because cfront does not check the accessibility of types, access errors for types are issued as warnings instead of errors.

- When matching arguments of an overloaded function, a **const** variable with value zero is not considered to be a null pointer constant. In general, in overload resolution a null pointer constant must be spelled "0" to be considered a null pointer constant (e.g., '\0' is not considered a null pointer constant).
- Inside the definition of a class type, the qualifier in the declarator for a member declaration is dropped if that qualifier names the class being defined.

```
struct S {
    void S::f();
};
```

- An alternate form of declaring pointer-to-member-function variables is supported, for example:

```
struct A {
    void f(int);
    static void sf(int);
    typedef void A::T3(int); // nonstd typedef decl
    typedef void T2(int);    // std typedef
};
typedef void A::T(int); // nonstd typedef decl
T* pmf = &A::f;         // nonstd ptr-to-member decl
A::T2* pf = A::sf;       // std ptr to static mem decl
A::T3* pmf2 = &A::f;     // nonstd ptr-to-member decl
```

where **T** is construed to name a routine type for a non-static member function of class **A** that takes an **int** argument and returns **void**; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of **T** and **pmf** in combination are equivalent to a single standard pointer-to-member declaration:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of **T**, is normally invalid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as **A::T3**, this feature changes the meaning of a valid declaration. cfront version 2.1 accepts declarations, such as **T**, even when **A** is an incomplete type; so this case is also excepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:


```

class B { protected: int i; };
class D : public B { void mf(); };
void D::mf() {
    int B::* pm1 = &B::i; // error, OK in cfront mode
    int D::* pm2 = &D::i; // OK
}

```



Protected member access checking for other operations (i.e., everything except taking a pointer-to-member address) is done in the normal manner.

- The destructor of a derived class may implicitly call the private destructor of a base class. In default mode this is an error but in cfront mode it is reduced to a warning. For example:

```

class A {
    ~A();
};
class B : public A {
    ~B();
};
B::~B(){} // Error except in cfront mode

```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword(identifier...)* is treated as an argument. For example:

```

class A { A(); };
double d;
A x(int(d));
A(x2);

```

By default `int(d)` is interpreted as a parameter declaration (with redundant parentheses), and so `x` is a function; but in cfront-compatibility mode `int(d)` is an argument and `x` is a variable.

The declaration `A(x2);` is also misinterpreted by cfront. It should be interpreted as the declaration of an object named `x2`, but in cfront mode is interpreted as a function style cast of `x2` to the type `A`.

Similarly, the declaration

```
int xyz(int());
```

declares a function named `xzy`, that takes a parameter of type "function taking no arguments and returning an `int`". In cfront mode this is interpreted as a declaration of an object that is initialized with the value `int()` (which evaluates to zero).

- A named bit-field may have a size of zero. The declaration is treated as though no name had been declared.
- Plain bit fields (i.e., bit fields declared with a type of `int`) are always unsigned.
- The name given in an elaborated type specifier is permitted to be a `typedef` name that is the synonym for a class name, e.g.,

```
typedef class A T;
class T *pa;           // No error in cfront
mode
```

- No warning is issued on duplicate size and sign specifiers.

```
short short int i;    // No warning in cfront mode
```

- Virtual function table pointer update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further-derived class. For example:

```
struct A {
    virtual void f() {}
    A() {}
    ~A() {}
};
struct B : public A {
    B() {}
    ~B() {f();}           // Should call A::f according to
                        // ARM 12.7
};
struct C : public B {
    void f() {}
} c;
```

In cfront compatibility mode, `B::~~B` calls `C::f`.

- An extra comma is allowed after the last argument in an argument list, as for example in

```
f(1, 2, );
```

- A constant pointer-to-member-function may be cast to a pointer-to-function. A warning is issued.

```

struct A {int f();};
main () {
    int (*p)();
    p = (int (*)( ))A::f;  // Okay, with warning
}

```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (i.e., like C structures), and the destructor is not called on the "copy". In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Note that because the argument is passed differently (by value instead of by address), code like this compiled in cfront mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.
- A union member may be declared to have the type of a class for which you have defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.
- When an unnamed class appears in a **typedef** declaration, the **typedef** name may appear as the class name in an elaborated type specifier.

```

typedef struct { int i, j; } S;
struct S x;  // No error in cfront mode

```

- Two member functions may be declared with the same parameter types when one is static and the other is non-static with a function qualifier.

```

class A {
    void f(int) const;
    static void f(int);  // No error in cfront mode
};

```

- The scope of a variable declared in the **for-init-statement** is the scope to which the **for** statement belongs.

```

int f(int i) {
    for (int j = 0; j < i; ++j) { /* ... */ }
    return j;  // No error in cfront mode
}

```

- Function types differing only in that one is declared **extern "C"** and the other **extern "C++"** can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
void f(PF);
void f(PCF);
```

`PF` and `PCF` are considered identical and `void f(PCF)` is treated as a compatible redeclaration of `f`. (By contrast, in standard C++ `PF` and `PCF` are different and incompatible types — `PF` is a pointer to an `extern "C++"` function whereas `PCF` is a pointer to an `extern "C"` function — and the two declarations of `f` create an overload set.)

- Functions declared `inline` have internal linkage.
- `enum` types are regarded as integral types.
- An uninitialized `const` object of non-POD class type is allowed even if its default constructor is implicitly declared:

```
struct A { virtual void f(); int i; };
const A a;
```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.
- If the user declares an `operator=` function in a class, but not one that can serve as the default `operator=`, and bitwise assignment could be done on the class, a default `operator=` is not generated; only the user-written `operator=` functions are considered for assignments (and therefore bitwise assignment is not done).
- A member function declaration whose return type is omitted (and thus implicitly `int`) and whose name is found to be that of a type is accepted if it takes no parameters:

```
typedef int I;

struct S {
    I(); // Accepted in Cfront mode (declares "int S::I()")
    I(int); // Not accepted
};
```

2.5 NAMESPACE SUPPORT

Namespaces are enabled by default except in the cfront modes. You can use the command-line options **--namespaces** and **--no-namespaces** to enable or disable the features.

Name lookup during template instantiations now does something that approximates the two-phase lookup rule of the standard. When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. The C++ compiler follows the new instantiation lookup rules for namespaces as closely as possible in the absence of a complete implementation of the new template name binding rules. Here is an example:

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}

namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);           // N::A<int>::f(int) calls
                              // N::g(int)
    int i2 = ai.f();           // N::A<int>::f() returns
                              // 0 (= N::x)

    N::A<double> ad;
    double d = ad.f(0);       // N::A<double>::f(double)
                              // calls M::g(double)
    double d2 = ad.f();       // N::A<double>::f() also
                              // returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.

- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for dependent function calls.

The lookup rules for overloaded operators are implemented as specified by the standard, which means that the operator functions in the global scope overload with the operator functions declared extern inside a function, instead of being hidden by them. The old operator function lookup rules are used when namespaces are turned off. This means a program can have different behavior, depending on whether it is compiled with namespace support enabled or disabled:

```
struct A { };
A operator+(A, double);
void f() {
    A a1;
    A operator+(A, int);
    a1 + 1.0;    // calls operator+(A, double)
                // with namespaces enabled but
}               // otherwise calls operator+(A, int);
```

2.6 TEMPLATE INSTANTIATION

The C++ language includes the concept of *templates*. A template is a description of a class or function that is a model for a family of related classes or functions.¹ For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create *instantiations* of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately, for several reasons:

- One would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows one to write a *specialization* of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (One could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it.
- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

(It should be noted that certain template entities are always instantiated when used, e.g., inline functions.)

¹ Since templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called **parameterized types**.

From these requirements, one can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. That is, the compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

This C++ compiler provides an instantiation mechanism that does automatic instantiation at link time. For cases where you want more explicit control over instantiation, the C++ compiler also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process.

2.6.1 AUTOMATIC INSTANTIATION

The goal of an automatic instantiation mode is to provide painless instantiation. You should be able to compile source files to object code, then link them and run the resulting program, and never have to worry about how the necessary instantiations get done.

In practice, this is hard for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses:

- AT&T/USL/Novell's *cfront* product saves information about each file it compiles in a special directory called **ptrepository**. It instantiates nothing during normal compilations. At link time, it looks for entities that are referenced but not defined, and whose mangled names indicate that they are template entities. For each such entity, it consults the **ptrepository** information to find the file containing the source for the entity, and it does a compilation of the source to generate an object file containing object code for that entity. This object code for instantiated objects is then combined with the "normal" object code in the link step.

If you are using *cf*ront you must follow a particular coding convention: all templates must be declared in `.h` files, and for each such file there must be a corresponding `.cc` file containing the associated definitions. The compiler is never told about the `.cc` files explicitly; one does not, for example, compile them in the normal way. The link step looks for them when and if it needs them, and does so by taking the `.h` filename and replacing its suffix.²

This scheme has the disadvantage that it does a separate compilation for each instantiated function (or, at best, one compilation for all the member functions of one class). Even though the function itself is often quite small, it must be compiled along with the declarations for the types on which the instantiation is based, and those declarations can easily run into many thousands of lines. For large systems, these compilations can take a very long time. The link step tries to be smart about recompiling instantiations only when necessary, but because it keeps no fine-grained dependency information, it is often forced to "recompile the world" for a minor change in a `.h` file. In addition, *cf*ront has no way of ensuring that preprocessing symbols are set correctly when it does these instantiation compilations, if preprocessing symbols are set other than on the command line.

- Borland's C++ compiler instantiates everything referenced in a compilation, then uses a special linker to remove duplicate definitions of instantiated functions.

If you are using Borland's compiler you must make sure that every compilation sees all the source code it needs to instantiate all the template entities referenced in that compilation. That is, one cannot refer to a template entity in a source file if a definition for that entity is not included by that source file. In practice, this means that either all the definition code is put directly in the `.h` files, or that each `.h` file includes an associated `.cc` (actually, `.cpp`) file.

This scheme is straightforward, and works well for small programs. For large systems, however, it tends to produce very large object files, because each object file must contain object code (and symbolic debugging information) for each template entity it references.

² The actual implementation allows for several different suffixes and provides a command-line option to change the suffixes sought.

Our approach is a little different. It requires that, for each instantiation required, there is some (normal, top-level, explicitly-compiled) source file that contains the definition of the template entity, a reference that causes the instantiation, and the declarations of any types required for the instantiation.³ This requirement can be met in various ways:

- The Borland convention: each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion: when the compiler sees a template declaration in a `.h` file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the *cfront* convention to be compiled with our approach. See the section on implicit inclusion.
- The ad hoc approach: you make sure that the files that define template entities also have the definitions of all the available types, and add code or pragmas in those files to request instantiation of the entities there.

Our compiler's automatic instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that *could* have been instantiated in each compilation. For any source file that makes use of a template instantiation an associated `.ii` file is created if one does not already exist (e.g., the compilation of `abc.cc` would result in the creation of `abc.ii`).
2. When the object files are linked together, a program called the *prelinker*, **prelk166**, is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.

³ Isn't this always the case? No. Suppose that file **A** contains a definition of class **X** and a reference to `Stack<X>::push`, and that file **B** contains the definition for the member function `push`. There would be no file containing both the definition of `push` and the definition of **X**.

3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in the associated instantiation request file (with, by default, a `.ii` suffix).
4. The prelinker then executes the compiler again to recompile each file for which the `.ii` file was changed. The original compilation command-line options (saved in the template information file) are used for the recompilation.
5. When the compiler compiles a file, it reads the `.ii` file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).
6. The prelinker repeats steps 3–5 until there are no more instantiations to be adjusted.
7. The object files are linked together.

Once the program has been linked correctly, the `.ii` files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the `.ii` files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. That's true even if the entire program is recompiled.

If you provide a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If you add a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper `.ii` file.

The `.ii` files should not, in general, require any manual intervention. One exception: if a definition is changed in such a way that some instantiation no longer compiles (it gets errors), and at the same time a specialization is added in another file, and the first file is being recompiled before the specialization file and is getting errors, the `.ii` file for the file getting the errors must be deleted manually to allow the prelinker to regenerate it.

If you supplied the **-v** option to the control program **cc166**, and the prelinker changes an instantiation assignment, the prelinker will issue messages like:

```
C++ prelinker: A<int>::f() assigned to file test.o
C++ prelinker: executing: cc166 -c test.cc
```

The automatic instantiation scheme can coexist with partial explicit control of instantiation by you through the use of pragmas or command-line specification of the instantiation mode. See the following sections.

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when "one instantiation per object" mode is specified, each instantiation is placed in its own object file. One-instantiation-per-object mode is useful when generating libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances. Without this feature it is necessary to create each individual instantiation object file using the manual instantiation mechanism.

The automatic instantiation mode is enabled by default. It can be turned off by the command-line option **--no-auto-instantiation**. If automatic instantiation is turned off, the extra information about template entities that could be instantiated in a file is not put into the object file.

2.6.2 INSTANTIATION MODES

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by a command line option:

--instantiate none

Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.

--instantiate used

Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.

--instantiate all

Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.

--instantiate local

Similar to **--instantiate used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions, so this is not suitable for production use. **--instantiate local** can not be used in conjunction with automatic template instantiation. If automatic instantiation **--instantiate local** option. If automatic instantiation is not enabled by default, use of **--instantiate local** and **--auto-instantiation** is an error.

In the case where the **cc166** command is given a single file to compile and link, e.g.,

```
cc166 test.cc
```

the compiler knows that all instantiations will have to be done in the single source file. Therefore, it uses the **--instantiate used** mode and suppresses automatic instantiation.

2.6.3 INSTANTIATION #PRAGMA DIRECTIVES

Instantiation pragmas can be used to control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

- The **instantiate** pragma causes a specified entity to be instantiated.
- The **do_not_instantiate** pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.

- The **can_instantiate** pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.

The argument to the instantiation pragma may be:

a template class name	<code>A<int></code>
a template class declaration	<code>class A<int></code>
a member function name	<code>A<int>::f</code>
a static data member name	<code>A<int>::i</code>
a static data declaration	<code>int A<int>::i</code>
a member function declaration	<code>void A<int>::f(int, char)</code>
a template function declaration	<code>char* f(int, float)</code>

A pragma in which the argument is a template class name (e.g., `A<int>` or `class A<int>`) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the **do_not_instantiate** pragma. For example,

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the **instantiate** pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T); // No body provided
template <class T> void g1(T); // No body provided
```

```

void f1(int) {} // Specific definition
void main()
{
    int      i;
    double   d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}

#pragma instantiate void f1(int)    // error - specific
                                   // definition
#pragma instantiate void g1(int)    // error - no body
                                   // provided

```

`f1(double)` and `g1(double)` will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (e.g., `A<int>::f`) can only be used as a pragma argument if it refers to a single user defined member function (i.e., not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```

#pragma instantiate char* A<int>::f(int, char*)

```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

2.6.4 IMPLICIT INCLUSION

When implicit inclusion is enabled, the C++ compiler is given permission to assume that if it needs a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.cc` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will look to see if a file `xyz.cc` exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity the C++ compiler needs to know the full path name of the file in which the template was declared and whether the file was included using the system include syntax (e.g., `#include <file.h>`). This information is not available for preprocessed source containing `#line` directives. Consequently, the C++ compiler will not attempt implicit inclusion for source code containing `#line` directives.

By default, the list of definition-file suffixes tried is `.cc`, `.cpp`, and `.cxx`. If `-c++` is supplied to the control program `cc166`, `.c` is also used as C++ file.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

The implicit inclusion mode can be turned on by the command-line option **`--implicit-include`**.

Implicit inclusions are only performed during the normal compilation of a file, (i.e., not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This may be done using the **`--no-preproc-only`** command line option. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files will appear as part of the preprocessed output in the precise location at which they were included in the compilation.

2.7 PREDEFINED MACROS

The C++ compiler defines a number of preprocessing macros. Many of them are only defined under certain circumstances. This section describes the macros that are provided and the circumstances under which they are defined.

All C predefined macros are also defined.

`__STDC__` Defined in ANSI C mode and in C++ mode. In C++ mode the value may be redefined. Not defined when embedded C++ is used.

`__FILE__` "current source filename"

`__LINE__` current source line number (int type)

`__TIME__` "hh:mm:ss"

`__DATE__` "Mmm dd yyyy"

`__MODEL` Identifies for which memory model the module is compiled.

`__cplusplus` Defined in C++ mode.

`c_plusplus` Defined in default C++ mode, but not in strict mode.

`__STDC_VERSION__`
Defined in ANSI C mode with the value 199409L. The name of this macro, and its value, are specified in Normative Addendum 1 of the ISO C Standard.

`__SIGNED_CHARS__`
Defined when plain `char` is signed. This is used in the `<limits.h>` header file to get the proper definitions of `CHAR_MAX` and `CHAR_MIN`.

`__WCHAR_T` Defined in C++ mode when `wchar_t` is a keyword.

`__BOOL` Defined in C++ mode when `bool` is a keyword.

`__ARRAY_OPERATORS`
Defined in C++ mode when array `new` and `delete` are enabled.

`__EXCEPTIONS`
Defined in C++ mode when exception handling is enabled.

- `__RTTI` Defined in C++ mode when RTTI is enabled.
- `__PLACEMENT_DELETE`
Defined in C++ mode when placement delete is enabled.
- `__NAMESPACES`
Defined in C++ mode when namespaces are supported (**--namespaces**).
- `__TSW_RUNTIME_USES_NAMESPACES`
Defined in C++ mode when the configuration flag `RUNTIME_USES_NAMESPACES` is TRUE. The name of this predefined macro is specified by a configuration flag. `__EDG_RUNTIME_USES_NAMESPACES` is the default.
- `__TSW_IMPLICIT_USING_STD`
Defined in C++ mode when the configuration flag `RUNTIME_USES_NAMESPACES` is TRUE and when the standard header files should implicitly do a using-directive on the `std` namespace (**--using-std**).
- `__TSW_CPP__`
Always defined.
- `__TSW_CPP_VERSION__`
Defined to an integral value that represents the version number of the C++ front end. For example, version 2.43 is represented as 243.
- `__embedded_cplusplus`
Defined as 1 in Embedded C++ mode.

2.8 PRECOMPILED HEADERS

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that **#include** them are relatively small. The C++ compiler provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation; then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the "snapshot point", verify that the corresponding precompiled header (PCH) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files can take a lot of disk space.

2.8.1 AUTOMATIC PRECOMPILED HEADER PROCESSING

When **--pch** appears on the command line, automatic precompiled header processing is enabled. This means the C++ compiler will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the "header stop" point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by **#pragma hdrstop** (see below) if that comes first. For example:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is **int** (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of **xxx.h** and **yyy.h**. If the first non-preprocessor token or the **#pragma hdrstop** appears within a **#if** block, the header stop point is the outermost enclosing **#if**. To illustrate, heres a more complicated example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again `int`, but the header stop point is the start of the `#if` block containing it. The PCH file will reflect the inclusion of `xxx.h` and conditionally the definition of `YYY_H` and inclusion of `yyy.h`; it will not contain the state produced by `#if TEST`.

A PCH file will be produced only if the header stop point and the code preceding it (mainly, the header files themselves) meet certain requirements:

- The header stop point must appear at file scope — it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

```
// xxx.h
class A {

// xxx.C
#include "xxx.h"
int i; };
```

- The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is `int`, but since it is not the start of a new declaration, no PCH file will be created:

```
// yyy.h
static

// yyy.C
#include "yyy.h"
int i;
```

- Similarly, the header stop point may not be inside a `#if` block or a `#define` started within a header file.

- The processing preceding the header stop must not have produced any errors. (Note: warnings and other diagnostics will not be reproduced when the PCH file is reused.)
- No references to predefined macros `__DATE__` or `__TIME__` may have appeared.
- No use of the `#line` preprocessing directive may have appeared.
- **#pragma no_pch** (see below) must not have appeared.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers. The minimum number of declarations required is 1.

When the host system does not support memory mapping, so that everything to be saved in the precompiled header file is assigned to preallocated memory (MS-Windows), two additional restrictions apply:

- The total memory needed at the header stop point cannot exceed the size of the block of preallocated memory.
- No single program entity saved can exceed 16384, the preallocation unit.

When a precompiled header file is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.
- The current directory (i.e., the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of preprocessing directives from the primary source file, including `#include` directives.
- The date and time of the header files specified in `#include` directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```
// a.cc
#include "xxx.h"
...                // Start of code
// b.cc
#include "xxx.h"
...                // Start of code
```

When **a.cc** is compiled with **--pch**, a precompiled header file named **a.pch** is created. Then, when **b.cc** is compiled (or when **a.cc** is recompiled), the prefix section of **a.pch** is read in for comparison with the current source file. If the command line options are identical, if **xxx.h** has not been modified, and so forth, then, instead of opening **xxx.h** and processing it line by line, the C++ compiler reads in the rest of **a.pch** and thereby establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for **xxx.h** and a second for **xxx.h** and **yyy.h**, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by an implementation-specified suffix (**pch** by default). Unless **--pch-dir** is specified (see below), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as

```
"test.cc": creating precompiled header file "test.pch"
```

is issued. The user may suppress the message by using the command-line option **--no-pch-messages**.

When the **--pch-verbose** option is used the C++ compiler will display a message for each precompiled header file that is considered that cannot be used giving the reason that it cannot be used.

In automatic mode (i.e., when **--pch** is used) the C++ compiler will deem a precompiled header file obsolete and delete it under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation; or
- if the precompiled header file has the same base name as the source file being compiled (e.g., **xxx.pch** and **xxx.cc**) but is not applicable for the current compilation (e.g., because of different command-line options).

This handles some common cases; other PCH file clean-up must be dealt with by other means (e.g., by the user).

Support for precompiled header processing is not available when multiple source files are specified in a single compilation: an error will be issued and the compilation aborted if the command line includes a request for precompiled header processing and specifies more than one primary source file.

2.8.2 **MANUAL PRECOMPILED HEADER PROCESSING**

Command-line option **--create-pch** *file-name* specifies that a precompiled header file of the specified name should be created.

Command-line option **--use-pch** *file-name* specifies that the indicated precompiled header file should be used for this compilation; if it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with **--pch-dir**, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The **--create-pch**, **--use-pch**, and **--pch** options may not be used together. If more than one of these options is specified, only the last one will apply. Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes — header stop points are determined the same way, PCH file applicability is determined the same way, and so forth.

2.8.3 OTHER WAYS TO CONTROL PRECOMPILED HEADERS

There are several ways in which the user can control and/or tune how precompiled headers are created and used.

- **#pragma hdrstop** may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. It enables you to specify where the set of header files subject to precompilation ends. For example,

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

Here, the precompiled header file will include processing state for **xxx.h** and **yyy.h** but not **zzz.h**. (This is useful if the user decides that the information added by what follows the **#pragma hdrstop** does not justify the creation of another PCH file.)

- **#pragma no_pch** may be used to suppress precompiled header processing for a given source file.
- Command-line option **--pch-dir** *directory-name* is used to specify the directory in which to search for and/or create a PCH file.

Moreover, when the host system does not support memory mapping and preallocated memory is used instead, then one of the command-line options **--pch**, **--create-pch**, or **--use-pch**, if it appears at all, must be the *first* option on the command line.

2.8.4 PERFORMANCE ISSUES

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, it does not cost much to write a precompiled header file out even if it does not end up being used, and if it *is* used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so one probably does not want many of them sitting around.

Thus, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, users should expect to reorder the `#include` sections of their source files and/or to group `#include` directives within a commonly used header file.

Below is an example of how this can be done. A common idiom is this:

```
#include "comnfile.h"
#pragma hdrstop
#include ...
```

where `comnfile.h` pulls in, directly and indirectly, a few dozen header files; the `#pragma hdrstop` is inserted to get better sharing with fewer PCH files. The PCH file produced for `comnfile.h` can be a bit over a megabyte in size. Another idiom, used by the source files involved in declaration processing, is this:

```
#include "comnfile.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

`decl_hdrs.h` pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the source files of a particular program can share just a few precompiled header files. If disk space were at a premium, you could decide to make `comnfile.h` pull in *all* the header files used — then, a single PCH file could be used in building the program.

Different environments and different projects will have different needs, but in general, users should be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to source code.

2.9 PROHIBITED C166 OPTIMIZATIONS

This section describes additional remarks which apply when compiling the C166/ST10 C code generated by **cp166** using **c166**.

2.9.1 'MAIN' LABELS IN A C++ APPLICATION

The symbol `'__main'` is generated by **c166** when compiling the `main()` function coded by the user in the application.

The symbol `'__main'` is generated by **cp166** when the `__main()` function is called. This function is part of the C++ library and is automatically called by the `main()` function for C++ initialization purposes. The function `'__main_called_more_than_once()'` is used to prevent multiple instantiations of the `__main()` function, thus preventing multiple initializations.

2.9.2 PROHIBITED C166 OPTIMIZATIONS

The **-Ot** optimization cannot be used when compiling the C166/ST10 C code generated by **cp166**.

With **-Ot**, tentative declarations (such as `int i;`) are turned into defining occurrences (e.g. `int i=0;`). This mechanism is used by **cp166** and will therefore generate an error when compiled with **c166** using the **-Ot** option.

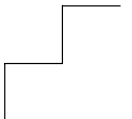
CHAPTER

3

COMPILER USE



TASKING



3

CHAPTER

3.1 INVOCATION

The invocation syntax of the C++ compiler is:

```
cp166 [option]... file
```



When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as '(' and '?') must be enclosed with " " or escaped. The **-?** option (in the C-shell) becomes: **"-?"** or **-\\?**.

The C++ compiler accepts a C++ source file name and command line options in random order. A C++ source file must have a **.cc**, **.cxx** or **.cpp** suffix.

Command line options may be specified using either single character option codes (e.g., **-A**), or keyword options (e.g., **--strict**). If an option requires an argument, the argument may immediately follow the option letter, or may be separated from the option letter by white space. A keyword option specification consists of two hyphens followed by the option keyword (e.g., **--strict**). Keyword options may be abbreviated by specifying as many of the leading characters of the option name as are needed to uniquely identify an option name (for example, the **--wchar_t-keyword** option may be abbreviated as **--wc**). Note that this is not supported by the control program! If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by **=option**. When the second form is used there may not be any white space on either side of the equals sign.

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The **-D** and **-U** options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the **--gen-c-file-name** option.

A summary of the options is given below. The next section describes the options in more detail.

Option	Description
-?	Display invocation syntax
--alternative-tokens	Enable or disable recognition of alternative tokens
--no-alternative-tokens	

Option	Description
--anachronisms --no-anachronisms	Enable or disable anachronisms
--arg-dep-lookup --no-arg-dep-lookup	Perform argument dependent lookup of unqualified function names
--array-new-and-delete --no-array-new-and-delete	Enable or disable support for array new and delete
--auto-instantiation --no-auto-instantiation -T	Enable or disable automatic instantiation of templates
--base-assign-op-is-default --no-base-assign-op-is-default	Enable or disable the anachronism of accepting a copy assignment operator with a base class as a default for the derived class
--bool --no-bool	Enable or disable recognition of <code>bool</code>
--brief-diagnostics --no-brief-diagnostics	Enable or disable a shorter form of diagnostic output
--cfront-2.1 -b	Compile C++ compatible with cfront version 2.1
--cfront-3.0	Compile C++ compatible with cfront version 3.0
--class-name-injection --no-class-name-injection	Add class name to the scope of the class
--comments -C	Keep comments in the preprocessed output
--const-string-literals --no-const-string-literals	Make string literals <code>const</code>
--create-pch file	Create a precompiled header file with the specified name
--define macro[(parm-list)] [=def] -Dmacro[(parm-list)][=def]	Define preprocessor <i>macro</i>

Option	Description
--dependencies	
-M	Preprocess only. Emit dependencies for make
--diag-suppress <i>tag[,tag]...</i>	
--diag-remark <i>tag[,tag]...</i>	
--diag-warning <i>tag[,tag]...</i>	
--diag-error <i>tag[,tag]...</i>	Override normal error severity
--display-error-number	Display error number in diagnostic messages
--distinct-template-signatures	
--no-distinct-template-signatures	Disallow or allow normal functions as template instantiation
--dollar	
-\$	Accept dollar signs in identifiers
--early-tiebreaker	Early handling of tie-breakers in overload resolution
--embedded	
--no-embedded	Enable or disable support for embedded C++ language extension keywords
--embedded-c++	Enable the diagnostics of noncompliance with the "Embedded C++" subset
--enum-overloading	
--no-enum-overloading	Enable or disable operator functions to overload builtin operators on enum-typed operands
--error-limit <i>number</i>	
-enumber	Specify maximum <i>number</i> of errors
--error-output <i>efile</i>	Send diagnostics to error list file
--exceptions	
--no-exceptions	
-x	Enable or disable support for exception handling
--explicit	
--no-explicit	Enable or disable support for the <code>explicit</code> specifier on constructor declarations

Option	Description
--extended-variadic-macros --no-extended-variadic-macros	Allow (or disallow) macros with a variable number of arguments and allow the naming of the list
--extern-inline --no-extern-inline	Enable or disable inline function with external C++ linkage
-f file	Read command line arguments from <i>file</i>
--force-vtbl	Force definition of virtual function tables
--for-init-diff-warning --no-for-init-diff-warning	Enable or disable warning when old-style <code>for</code> -scoping is used
--friend-injection --no-friend-injection	Control the visibility of friend declarations
--gen-c-file-name file -o file	Specify name of generated C output <i>file</i>
--guiding-decls --no-guiding-decls	Enable or disable recognition of "guiding declarations" of template functions
--implicit-extern-c-type-conversion --no-implicit-extern-c-type-conversion	Enable or disable implicit type conversion between external C and C++ function pointers
--implicit-include --no-implicit-include -B	Enable or disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
--implicit-typename --no-implicit-typename	Enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype
--incl-suffixes suffixes	Set the valid suffixes for include files

Option	Description
--include-directory <i>dir</i> -I <i>dir</i>	Look in directory <i>dir</i> for include files
--include-file <i>file</i>	Include <i>file</i> at the beginning of the compilation
--inlining --no-inlining	Enable or disable minimal inlining of function calls
--instantiate <i>mode</i> -t <i>mode</i>	Control instantiation of external template entities
--instantiation-dir <i>dir</i>	Write instantiation files to <i>dir</i>
--late-tiebreaker	Late handling of tie-breakers in overload resolution
--list-file <i>lfile</i> -L <i>lfile</i>	Generate raw list file <i>lfile</i>
--long-lifetime-temps --short-lifetime-temps	Select lifetime for temporaries
--long-preserving-rules --no-long-preserving-rules	Enable or disable K&R arithmetic conversion rules for longs
-M [<i>t s m l h</i>]	Select memory model: tiny, small, medium, large or huge
--namespaces --no-namespaces	Enable or disable the support for namespaces
--new-for-init	New-style <code>for</code> -scoping rules
--no-code-gen -n	Do syntax checking only
--no-line-commands -P	Preprocess only. Remove line control information and comments
--nonconst-ref-anachronism --no-nonconst-ref-anachronism	Enable or disable the anachronism of allowing a reference to <code>nonconst</code> to bind to a class rvalue of the right type
--nonstd-qualifier-deduction --no-nonstd-qualifier-deduction	Use (or do not use) a non-standard template argument deduction method

Option	Description
--nonstd-using-decl --no-nonstd-using-decl	Allow or disallow unqualified name in non-member using declaration
--no-preproc-only	Specify that a full compilation should be done (not just preprocessing)
--no-use-before-set-warnings -j	Suppress warnings on local automatic variables that are used before their values are set
--no-warnings -w	Suppress all warning messages
--old-for-init	Old-style <code>for</code> -scoping rules
--old-line-commands	Put out line control information in the form <code># nnn</code> instead of <code>#line nnn</code>
--old-specializations --no-old-specializations	Enable or disable old-style template specialization
--old-style-preprocessing	Forces pcc style preprocessing
--one-instantiation-per-object	Create separate instantiation files
--output file	Write preprocess output in <i>file</i>
--pch	Automatically use and/or create a precompiled header file
--pch-dir dir	Specify directory <i>dir</i> in which to search for and/or create a precompiled header file
--pch-messages --no-pch-messages	Enable or disable the display of a message indicating that a precompiled header file was created or used in the current compilation
--pch-verbose	Generate a message when a precompiled header file cannot be used
--pending-instantiations n	Maximum number of instantiations for a single template (default 64)
--preprocess -E	Preprocess only. Keep line control information and remove comments
--remarks -r	Issue remarks

Option	Description
--remove-unneeded-entities --no-remove-unneeded-entities	Enable or disable the removal of unneeded entities from the generated intermediate C file
--rtti --no-rtti	Enable or disable support for RTTI (run-time type information)
--signed-chars -s	Treat all 'char' variables as signed
--special-subscript-cost --no-special-subscript-cost	Enable or disable a special nonstandard weighting of the conversion to the integral operand of the [] operator in overload resolution.
--strict -A	Strict ANSI C++. Issue errors on non-ANSI features
--strict-warnings -a	Strict ANSI C++. Issue warnings on non-ANSI features
--suppress-typeinfo-vars	Suppress type info variables in generated C
--suppress-vtbl	Suppress definition of virtual function tables
--sys-include <i>dir</i>	Look in directory <i>dir</i> for system include files
--timing -#	Generate compilation timing information
--trace-includes -H	Preprocess only. Generate list of included files
--tsw-diagnostics --no-tsw-diagnostics	Enable or disable TASKING style diagnostic messages
--typename --no-typename	Enable or disable recognition of <code>typename</code>
--undefine <i>macro</i> -U<i>macro</i>	Remove preprocessor <i>macro</i>

Option	Description
--unsigned-chars -u	Treat all 'char' variables as unsigned
--use-pch file	Use a precompiled header file of the specified name
--using-std --no-using-std	Enable or disable implicit use of the std namespace when standard header files are included
--variadic-macros --no-variadic-macros	Allow (or disallow) macros with a variable number of arguments
--version -V -v	Display version header only
--wchar_t-keyword --no-wchar_t-keyword	Enable or disable recognition of wchar_t as a keyword
--wrap-diagnostics --no-wrap-diagnostics	Enable or disable wrapping of diagnostic messages
--xref xfile -X xfile	Generate cross-reference file xfile

Table 3-1: Compiler options (alphabetical)

Description	Option
Include options	
Look in <i>dir</i> for include files	--include-directory dir -I dir
Look in <i>dir</i> for system include files	--sys-include dir
Set the valid suffixes for include files	--incl-suffixes suffixes
Include <i>file</i> at the beginning of the compilation	--include-file file
Read command line arguments from <i>file</i>	-f file

Description	Option
Preprocess options	
Preprocess only. Keep line control information and remove comments	--preprocess -E
Preprocess only. Remove line control information and comments	--no-line-commands -P
Keep comments in the preprocessed output	--comments -C
Do syntax checking only	--no-code-gen -n
Specify that a full compilation should be done (not just preprocessing)	--no-preproc-only
Put out line control information in the form <i># nnn</i> instead of <i>#line nnn</i>	--old-line-commands
Forces pcc style preprocessing	--old-style-preprocessing
Preprocess only. Emit dependencies for make	--dependencies -M
Preprocess only. Generate list of included files	--trace-includes -H
Define preprocessor <i>macro</i>	--define macro[(<i>parm-list</i>)] [=def] -Dmacro[(<i>parm-list</i>)] [=def]
Remove preprocessor <i>macro</i>	--undefine macro -Umacro
Allow (or disallow) macros with a variable number of arguments	--variadic-macros --no-variadic-macros
Allow (or disallow) macros with a variable number of arguments and allow the naming of the list	--extended-variadic-macros --no-extended-variadic-macros
Language control options	
Strict ANSI C++. Issue errors on non-ANSI features	--strict -A
Strict ANSI C++. Issue warnings on non-ANSI features	--strict-warnings -a
Select memory model: tiny, small, medium, large or huge	-M[t s m l h]
Compile C++ compatible with cfront version 2.1	--cfront-2.1 -b
Compile C++ compatible with cfront version 3.0	--cfront-3.0

Description	Option
Accept dollar signs in identifiers	--dollar -\$
Treat all 'char' variables as signed	--signed-chars -s
Treat all 'char' variables as unsigned	--unsigned-chars -u
Enable or disable K&R arithmetic conversion rules for longs	--long-preserving-rules --no-long-preserving-rules
Make string literals const	--const-string-literals --no-const-string-literals
Enable or disable support for exception handling	--exceptions --no-exceptions -x
Enable the diagnostics of noncompliance with the "Embedded C++" subset	--embedded-c++
Enable or disable support for embedded C++ language extension keywords	--embedded --no-embedded
Enable or disable operator functions to overload builtin operators on enum-typed operands	--enum-overloading --no-enum-overloading
Enable or disable support for the <code>explicit</code> specifier on constructor declarations	--explicit --no-explicit
Enable or disable inline function with external C++ linkage	--extern-inline --no-extern-inline
Enable or disable implicit type conversion between external C and C++ function pointers	--implicit-extern-c-type-conversion --no-implicit-extern-c-type-conversion
Suppress type info variables in generated C	--suppress-typeinfo-vars
Suppress definition of virtual function tables	--suppress-vtbl
Force definition of virtual function tables	--force-vtbl
Enable or disable anachronisms	--anachronisms --no-anachronisms

Description	Option
Enable or disable the anachronism of accepting a copy assignment operator with a base class as a default for the derived class	--base-assign-op-is-default --no-base-assign-op-is-default
Enable or disable the anachronism of allowing a reference to nonconst to bind to a class rvalue of the right type	--nonconst-ref-anachronism --no-nonconst-ref-anachronism
Use (or do not use) a non-standard template argument deduction method	--nonstd-qualifier-deduction --no-nonstd-qualifier-deduction
Allow or disallow unqualified name in non-member using declaration	--nonstd-using-decl --no-nonstd-using-decl
Perform argument dependent lookup of unqualified function names	--arg-dep-lookup --no-arg-dep-lookup
Add class name to the scope of the class	--class-name-injection --no-class-name-injection
Control the visibility of friend declarations	--friend-injection --no-friend-injection
Early or late handling of tie-breakers in overload resolution	--early-tiebreaker --late-tiebreaker
Enable or disable support for array new and delete	--array-new-and-delete --no-array-new-and-delete
Enable or disable support for namespaces	--namespaces --no-namespaces
New-style for-scoping rules	--new-for-init
Old-style for-scoping rules	--old-for-init
Enable or disable implicit use of the std namespace when standard header files are included	--using-std --no-using-std
Enable or disable support for RTTI (run-time type information)	--rtti --no-rtti
Enable or disable recognition of bool	--bool --no-bool
Enable or disable recognition of typename	--typename --no-typename
Enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype	--implicit-typename --no-implicit-typename

Description	Option
Enable or disable a special nonstandard weighting of the conversion to the integral operand of the [] operator in overload resolution.	--special-subscript-cost --no-special-subscript-cost
Enable or disable recognition of <code>wchar_t</code> as a keyword	--wchar_t-keyword --no-wchar_t-keyword
Select lifetime for temporaries	--long-lifetime-temps --short-lifetime-temps
Enable or disable recognition of alternative tokens	--alternative-tokens --no-alternative-tokens
Enable or disable minimal inlining of function calls	--inlining --no-inlining
Enable or disable the removal of unneeded entities from the generated intermediate C file	--remove-unneeded-entities --no-remove-unneeded-entities
Template instantiation options	
Control instantiation of external template entities	--instantiate mode -t mode
Enable or disable automatic instantiation of templates	--auto-instantiation --no-auto-instantiation -T
Create separate instantiation files	--one-instantiation-per-object
Write instantiation files to <i>dir</i>	--instantiation-dir dir
Enable or disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated	--implicit-include --no-implicit-include -B
Maximum number of instantiations for a single template (default 64)	--pending-instantiations n
Dis-allow or allow normal functions as template instantiation	--distinct-template-signatures --no-distinct-template-signatures
Enable or disable recognition of "guiding declarations" of template functions	--guiding-decls --no-guiding-decls
Enable or disable old-style template specialization	--old-specializations --no-old-specializations

Description	Option
Precompiled header options	
Automatically use and/or create a precompiled header file	--pch
Create a precompiled header file with the specified name	--create-pch file
Use a precompiled header file of the specified name	--use-pch file
Specify directory <i>dir</i> in which to search for and/or create a precompiled header file	--pch-dir dir
Enable or disable the display of a message indicating that a precompiled header file was created or used in the current compilation	--pch-messages --no-pch-messages
Generate a message when a precompiled header file cannot be used	--pch-verbose
Output file options	
Write preprocess output in <i>file</i>	--output file
Specify name of generated C output <i>file</i>	--gen-c-file-name file -o file
Diagnostic options	
Display invocation syntax	-?
Display version header only	--version -V -v
Generate compilation timing information	--timing -#
Send diagnostics to error list file	--error-output efile
Generate raw list file <i>lfile</i>	--list-file lfile -L lfile
Generate cross-reference file <i>xfile</i>	--xref xfile -X xfile
Override normal error severity	--diag-suppress tag[,tag]... --diag-remark tag[,tag]... --diag-warning tag[,tag]... --diag-error tag[,tag]...
Display error number in diagnostic messages	--display-error-number

Description	Option
Specify maximum <i>number</i> of errors	--error-limit <i>number</i> -<i>enumber</i>
Issue remarks	--remarks -r
Suppress all warning messages	--no-warnings -w
Suppress warnings on local automatic variables that are used before their values are set	--no-use-before-set-warnings -j
Enable or disable a shorter form of diagnostic output	--brief-diagnostics --no-brief-diagnostics
Enable or disable TASKING style diagnostic messages	--tsw-diagnostics --no-tsw-diagnostics
Enable or disable wrapping of diagnostic messages	--wrap-diagnostics --no-wrap-diagnostics
Enable or disable warning when old-style <code>for</code> -scoping is used	--for-init-diff-warning --no-for-init-diff-warning

Table 3-2: Compiler options (functional)

3.1.1 DETAILED DESCRIPTION OF THE COMPILER OPTIONS

Option letters are listed below. If the same option is used more than once, the first (most left) occurrence is used. The placement of command line options is of no importance except for the **-I** option. Some options also have a "no-" form. These options are described together.

-?

Option:

-?

Description:

Display an explanation of options at stdout.

Example:

cp166 -?

--alternative-tokens

Option:

--alternative-tokens
--no-alternative-tokens

Default:

--alternative-tokens

Description:

Enable or disable recognition of alternative tokens. This controls recognition of the digraph tokens in C++, and controls recognition of the operator keywords (e.g., **not**, **and**, **bitand**, etc.).

Example:

To disable operator keywords (e.g., "not", "and") and digraphs, enter:

```
cp166 --no-alternative-tokens test.cc
```

--anachronisms

Option:

--anachronisms
--no-anachronisms

Default:

--no-anachronisms

Description:

Enable or disable anachronisms.

Example:

```
cp166 --anachronisms test.cc
```



--nonconst-ref-anachronisms,
--cfront-2.1 / -b / --cfront-3.0

Section *Anachronisms Accepted* in chapter *Language Implementation*.

--arg-dep-lookup

Option:

--arg-dep-lookup
--no-arg-dep-lookup

Default:

--arg-dep-lookup

Description:

Controls whether argument dependent lookup of unqualified function names is performed.

Example:

```
cp166 --no-arg-dep-lookup test.cc
```

--array-new-and-delete

Option:

--array-new-and-delete
--no-array-new-and-delete

Default:

--array-new-and-delete

Description:

Enable or disable support for array new and delete.

Example:

```
cp166 --no-array-new-and-delete test.cc
```


--auto-instantiation / -T

Option:

-T / --auto-instantiation
--no-auto-instantiation

Default:

--auto-instantiation

Description:

-T is equivalent to **--auto-instantiation**. Enable or disable automatic instantiation of templates.

Example:

```
cp166 --no-auto-instantiation test.cc
```



--instantiate / -t

Section *Template Instantiation* in chapter *Language Implementation*.

--base-assign-op-is-default

Option:

--base-assign-op-is-default
--no-base-assign-op-is-default

Default:

--base-assign-op-is-default (in cfront compatibility mode)

Description:

Enable or disable the anachronism of accepting a copy assignment operator that has an input parameter that is a reference to a base class as a default copy assignment operator for the derived class.

Example:

```
cp166  --base-assign-op-is-default test.cc
```

--bool

Option:

--bool
--no-bool

Default:

--bool

Description:

Enable or disable recognition of the `bool` keyword.

Example:

```
cp166 --no-bool test.cc
```

--brief-diagnostics

Option:

--brief-diagnostics
--no-brief-diagnostics

Default:

--brief-diagnostics

Description:

Enable or disable a mode in which a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

Example:

```
cp166 --no-brief-diagnostics test.cc
```



--wrap-diagnostics

Chapter *Compiler Diagnostics* and Appendix *Error Messages*.

--cfront-version / -b

Option:

-b / --cfront-2.1
--cfront-3.0

Default:

Normal C++ mode.

Description:

-b is equivalent to **--cfront-2.1**. **--cfront-2.1** or **--cfront-3.0** enable compilation of C++ with compatibility with cfront version 2.1 or 3.0 respectively. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront) release 2.1 or 3.0 respectively. These options also enable acceptance of anachronisms.

Example:

To compile C++ compatible with cfront version 3.0, enter:

```
cp166 --cfront-3.0 test.cc
```



--anachronisms

Section *Extensions Accepted in Cfront 2.1 and 3.0 Compatibility Mode* in chapter *Language Implementation*.

--class-name-injection

Option:

--class-name-injection
--no-class-name-injection

Default:

--class-name-injection

Description:

Controls whether the name of a class is injected into the scope of the class (as required by the standard) or is not injected (as was true in earlier versions of the C++ language).

Example:

```
cp166  --no-class-name-injection test.cc
```

--comments / -C

Option:

-C
--comments

Description:

Keep comments in the preprocessed output. This should be specified after either **--preprocess** or **--no-line-commands**; it does not of itself request preprocessing output.

Example:

To do preprocessing only, with comments and with line control information, enter:

```
cp166 -E -C test.cc
```



```
--preprocess / -E, --no-line-commands / -P
```

--const-string-literals

Option:

--const-string-literals
--no-const-string-literals

Default:

--const-string-literals

Description:

Control whether C++ string literals and wide string literals are **const** (as required by the standard) or non-**const** (as was true in earlier versions of the C++ language).

Example:

```
cp166  --no-const-string-literals test.cc
```


--create-pch

Option:

--create-pch *filename*

Arguments:

A filename specifying the precompiled header file to create.

Description:

If other conditions are satisfied (see the *Precompiled Headers* section), create a precompiled header file with the specified name. If **--pch** (automatic PCH mode) or **--use-pch** appears on the command line following this option, its effect is erased.

Example:

To create a precompiled header file with the name `test.pch`, enter:

```
cp166  --create-pch test.pch test.cc
```



--pch, --use-pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--define / -D

Option:

-D*macro* [(*parm-list*)] [= *def*]
--define *macro* [(*parm-list*)] [= *def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* to the preprocessor, as in #define. If *def* is not given ('=' is absent), '1' is assumed. Function-style macros can be defined by appending a macro parameter list to *name*. Any number of symbols can be defined. The definition can be tested by the preprocessor with #if, #ifdef and #ifndef, for conditional compilations.

Example:

```
cp166 -DNORAM -DPI=3.1416 test.cc
```



--undefine / -U

--dependencies / -M

Option:

-M
--dependencies

Description:

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of dependency lines suitable for input to a 'make' utility.



When implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies unless **--no-preproc-only** is also used.



When you use the control program you have to use the **-Em** option instead, to obtain the same result.

Examples:

```
cp166 -M test.cc
```

```
test.ic: test.cc
```



```
--preprocess / -E, --no-line-commands / -P
```

--diag-option

Option:

- diag-suppress** *tag[,tag]...*
- diag-remark** *tag[,tag]...*
- diag-warning** *tag[,tag]...*
- diag-error** *tag[,tag]...*

Arguments:

A mnemonic error tag or an error number.

Description:

Override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number. The error tag names and error numbers are listed in the *Error Messages* appendix.

Example:

When you want diagnostic error 20 to be a warning, enter:

```
cp166 --diag-warning 20 test.cc
```



Chapter *Compiler Diagnostics* and Appendix *Error Messages*.

--display-error-number

Option:

--display-error-number

Description:

Display the error message number in any diagnostic messages that are generated. The option may be used to determine the error number to be used when overriding the severity of a diagnostic message. The error numbers are listed in the *Error Messages* appendix.

Normally, diagnostics are written to stderr in the following form:

"filename", line line_num: message

With **--display-error-number** this form will be:

"filename", line line_num: severity #err_num: message

or:

"filename", line line_num: severity #err_num-D: message

If the severity may be overridden, the error number will include the suffix **-D** (for discretionary); otherwise no suffix will be present.

Example:

```
cp166 --display-error-number test.cc
```

```
"test.cc", line 7: error #64-D: declaration does not
    declare anything
```

```
    struct ;
    ^
```



Chapter *Compiler Diagnostics* and Appendix *Error Messages*.

--distinct-template-signatures

Option:

--distinct-template-signatures
--no-distinct-template-signatures

Default:

--distinct-template-signatures

Description:

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. The default is **--distinct-template-signatures**, under which a normal function cannot be used to satisfy the need for a template instance; e.g., a function "void f(int)" could not be used to satisfy the need for an instantiation of a template "void f(T)" with T set to int.

--no-distinct-template-signatures provides the older language behavior, under which a non-template function can match a template function. Also controls whether function templates may have template parameters that are not used in the function signature of the function template

Example:

```
cp166  --no-distinct-template-signatures test.cc
```

--dollar / -\$

Option:

-\$
--dollar

Default:

No dollar signs are allowed in identifiers.

Description:

Accept dollar signs in identifiers. Names like **A\$VAR** are allowed.

Example:

cp166 -\$ test.cc

--early-tiebreaker / --late-tiebreaker

Option:

--early-tiebreaker
--late-tiebreaker

Default:

--early-tiebreaker

Description:

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach). In "late" tie-breaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

Example:

```
cp166 --late-tiebreaker test.cc
```


--embedded

Option:

--embedded
--no-embedded

Default:

--embedded

Description:

Enable or disable support for embedded C++ language extension keywords.

Example:

To disable embedded C++ language extension keywords, enter:

```
cp166 --no-embedded test.cc
```

--embedded-c++

Option:

--embedded-c++

Description:

Enable the diagnostics of noncompliance with the “Embedded C++” subset (from which templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and **mutable** are excluded.

Example:

To enable the diagnostics of noncompliance with the “Embedded C++” subset, enter:

```
cp166 --embedded-c++ test.cc
```

--enum-overloading

Option:

--enum-overloading
--no-enum-overloading

Default:

--enum-overloading

Description:

Enable or disable support for using operator functions to overload builtin operations on enum-typed operands.

Example:

To disable overloading builtin operations on enum-typed operands, enter:

```
cp166 --no-enum-overloading test.cc
```

--error-limit / -e

Option:

-*enumber***
--error-limit *number*

Arguments:

An error limit number.

Default:

--error-limit 100

Description:

Set the error limit to *number*. The C++ compiler will abandon compilation after this number of errors (remarks and warnings are not counted toward the limit). By default, the limit is 100.

Example:

When you want compilation to stop when 10 errors occurred, enter:

```
cp166 -e10 test.cc
```

--error-output

Option:

--error-output *efile*

Arguments:

The name for an error output file.

Description:

Redirect the output that would normally go to stderr (that is, diagnostic messages) to the file *efile*. This option is useful on systems where output redirection of files is not well supported. If used, this option should probably be specified first in the command line, since otherwise any command-line errors for options preceding the **--error-output** would be written to stderr before redirection.

Example:

To write errors to the file `test.err` instead of stderr, enter:

```
cp166 --error-output test.err test.cc
```

--exceptions / -x

Option:

-x / --exceptions
--no-exceptions

Default:

--no-exceptions

Description:

Enable or disable support for exception handling. **-x** is equivalent to **--exceptions**.

Example:

```
cp166 --exceptions test.cc
```

--explicit

Option:

--explicit
--no-explicit

Default:

--explicit

Description:

Enable or disable support for the `explicit` specifier on constructor declarations.

Example:

To disable support for the `explicit` specifier on constructor declarations, enter:

```
cp166 --no-explicit test.cc
```

--extended-variadic-macros

Option:

--extended-variadic-macros
--no-extended-variadic-macros

Default:

--no-extended-variadic-macros

Description:

Allow or disallow macros with a variable number of arguments (implies **--variadic-macros**) and allow or disallow the naming of the variable argument list.

Example:

```
cp166 --extended-variadic-macros test.cc
```



--variadic-macros

--extern-inline

Option:

--extern-inline
--no-extern-inline

Default:

--extern-inline

Description:

Enable or disable support for `inline` functions with external linkage in C++. When `inline` functions are allowed to have external linkage (as required by the standard), then `extern` and `inline` are compatible specifiers on a non-member function declaration; the default linkage when `inline` appears alone is external (that is, `inline` means `extern inline` on non-member functions); and an `inline` member function takes on the linkage of its class (which is usually external). However, when `inline` functions have only internal linkage (as specified in the ARM), then `extern` and `inline` are incompatible; the default linkage when `inline` appears alone is internal (that is, `inline` means `static inline` on non-member functions); and `inline` member functions have internal linkage no matter what the linkage of their class.

Example:

```
cp166 --no-extern-inline test.cc
```

-f

Option:

-f *filename*

Arguments:

The name of an option file.

Description:

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the C++ compiler.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **-f** multiple times.

Format of an option file:

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

```
"This has a single quote ' embedded"
```

```
'This has a double quote " embedded'
```

```
'This has a double quote " and \  
a single quote ''' embedded"
```

- When a text line reaches its length limit, use a '\ ' to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \  
line"
```

```
-> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `myoptions` contains the following lines:

```
-I/proj/include  
test.cc
```

Specify the option file to the C++ compiler:

```
cp166 -f myoptions
```

This is equivalent to the following command line:

```
cp166 -I/proj/include test.cc
```

--for-init-diff-warning

Option:

--for-init-diff-warning
--no-for-init-diff-warning

Default:

--for-init-diff-warning

Description:

Enable or disable a warning that is issued when programs compiled under the new for-init scoping rules would have had different behavior under the old rules. The diagnostic is only put out when the new rules are used.

Example:

```
cp166  --no-for-init-diff-warning test.cc
```



--new-for-init / --old-for-init

--force-vtbl

Option:

--force-vtbl

Description:

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. See **--suppress-vtbl**.

Example:

```
cp166 --force-vtbl test.cc
```



--suppress-vtbl

--friend-injection

Option:

--friend-injection
--no-friend-injection

Default:

--no-friend-injection

Description:

Controls whether the name of a class or function that is declared only in `friend` declarations is visible when using the normal lookup mechanisms. When `friend` names are injected, they are visible to such lookups. When `friend` names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

Example:

cp166 --friend-injection test.cc



--arg-dep-lookup

--gen-c-file-name / -o

Option:

-o *file*
--gen-c-file-name *file*

Arguments:

An output filename.

Default:

Module name with `.ic` suffix.

Description:

This option specifies the file name to be used for the generated C output.

Example:

To specify the file `out.ic` as the output file instead of `test.ic`, enter:

```
cp166 --gen-c-file-name out.ic test.cc
```

--guiding-decls

Option:

--guiding-decls
--no-guiding-decls

Default:

--guiding-decls

Description:

Enable or disable recognition of “guiding declarations” of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T) { ... }  
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise, it is an independent function for which a definition must be supplied. If **--no-guiding-decls** is combined with **--old-specializations**, a specialization of a non-member template function is not recognized — it is treated as a definition of an independent function.

Example:

```
cp166 --no-guiding-decls test.cc
```



--old-specializations

--implicit-extern-c-type-conversion

Option:

--implicit-extern-c-type-conversion
--no-implicit-extern-c-type-conversion

Default:

--implicit-extern-c-type-conversion

Description:

Enable or disable an extension to permit implicit type conversion in C++ between a pointer to an **extern "C"** function and a pointer to an **extern "C++"** function. This extension is allowed in environments where C and C++ functions share the same calling conventions.

Example:

```
cp166 --no-implicit-extern-c-type-conversion test.cc
```

--implicit-include / -B

Option:

-B / --implicit-include
--no-implicit-include

Default:

--no-implicit-include

Description:

Enable or disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. **-B** is equivalent to **--implicit-include**.

Example:

```
cp166 --implicit-include test.cc
```



--instantiate / -t

Section *Template Instantiation* in chapter *Language Implementation*.

--implicit-typename

Option:

--implicit-typename
--no-implicit-typename

Default:

--implicit-typename

Description:

Enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype.

Example:

```
cp166 --no-implicit-typename test.cc
```



```
--typename
```

--incl-suffixes

Option:

--include-suffixes *suffixes*

Arguments:

A colon-separated list of suffixes (e.g., "h:hpp:").

Description:

Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. If a null suffix is to be allowed, it must be included in the suffix list.

The default suffix list is no extension, **.h** and **.hpp**.

Example:

To allow only the suffixes **.h** and **.hpp** as include file extensions, enter:

```
cp166 --incl-suffixes h:hpp test.cc
```



Section 3.2, *Include Files*.

--include-directory / -I

Option:

-I*directory*
--include-directory *directory*

Arguments:

The name of the directory to search for include file(s).

Description:

Change the algorithm for searching #include files whose names do not have an absolute pathname to look in *directory*.

Example:

```
cp166 -I/proj/include test.cc
```



Section 3.2, *Include Files*.

--sys-include

--include-file

Option:

--include-file *filename*

Arguments:

The name of the file to be included at the beginning of the compilation.

Description:

Include the source code of the indicated file at the beginning of the compilation. This can be used to establish standard macro definitions, etc.

The filename is searched for in the directories on the include search list.

Example:

```
cp166 --include-file extra.h test.cc
```



Section 3.2, *Include Files*.

--inlining

Option:

--inlining
--no-inlining

Default:

--inlining

Description:

Enable or disable minimal inlining of function calls.

Example:

To disable function call inlining, enter:

```
cp166 --no-inlining test.cc
```

--instantiate / -t

Option:

-t*mode*
--instantiate *mode*

Pragma:

instantiate *mode*

Arguments:

The instantiation mode, which can be one of:

none
used
all
local

Default:

-tnone

Description:

Control instantiation of external template entities. External template entities are external (that is, noninline and nonstatic) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition:

none	Instantiate no template entities. This is the default.
used	Instantiate only the template entities that are used in this compilation.
all	Instantiate all template entities whether or not they are used.
local	Instantiate only the template entities that are used in this compilation, and force those entities to be local to this compilation.

Example:

To specify to instantiate only the template entities that are used in this compilation, enter:

```
cp166 -tused test.cc
```



--auto-instantiation / -T

Section *Template Instantiation* in chapter *Language Implementation*.

--instantiation-dir

Option:

--instantiation-dir *directory*

Arguments:

The name of the directory to write instantiation files to.

Description:

You can use this option in combination with option

--one-instantiation-per-object to specify a directory into which the generated object files should be put.

Example:

To create separate instantiation files in directory `/proj/template`, enter:

```
cp166  --one-instantiation-per-object \  
       --instantiation-dir /proj/template test.cc
```



Section *Template Instantiation* in chapter *Language Implementation*.

--one-instantiation-per-object

--list-file / -L

Option:

-L*file*
--list-file *file*

Arguments:

The name of the list file.

Description:

Generate raw listing information in the file *file*. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the C++ compiler. Each line of the listing file begins with a key character that identifies the type of line, as follows:

- N**: a normal line of source; the rest of the line is the text of the line.
- X**: the expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the N line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications).
- S**: a line of source skipped by an `#if` or the like; the rest of the line is text. Note that the `#else`, `#elif`, or `#endif` that ends a skip is marked with an N.
- L**: an indication of a change in source position. The line has a format similar to the `#` line-identifying directive output by `cpp`, that is to say

L *line_number* "file-name" *key*

where *key* is,

- 1 for entry into an include file;
- 2 for exit from an include file;

and omitted otherwise.

The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for `#line` directives (*key* is omitted). L lines indicate the source position of the following source line in the raw listing file.

R, W, E, or C: an indication of a diagnostic (R for remark, W for warning, E for error, and C for catastrophic error). The line has the form

S "file-name" line_number column-number message-text

where *S* is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name ("") and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lower case version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example:

To write raw listing information to the file `test.lst`, enter:

```
cp166 -L test.lst test.cc
```

--long-lifetime-temps / --short-lifetime-temps

Option:

--long-lifetime-temps
--short-lifetime-temps

Default:

--long-lifetime-temps (cfront)
--short-lifetime-temps (standard C++)

Description:

Select the lifetime for temporaries: short means to end of full expression; long means to the earliest of end of scope, end of switch clause, or the next label. Short is standard C++, and long is what cfront uses (the cfront compatibility modes select long by default).

Example:

cp166 --long-lifetime-temps test.cc

--long-preserving-rules

Option:

--long-preserving-rules
--no-long-preserving-rules

Default:

--no-long-preserving-rules

Description:

Enable or disable the K&R usual arithmetic conversion rules with respect to `long`. This means the rules of K&R I, Appendix A, 6.6. The significant difference is in the handling of "`long op unsigned int`" when `int` and `long` are the same size. The ANSI/ISO rules say the result is `unsigned long`, but K&R I says the result is `long` (`unsigned long` did not exist in K&R I).

The default is the ANSI/ISO rule.

Example:

```
cp166 --long-preserving-rules test.cc
```

-Mmodel

Option:

-Mmodel

Arguments:

The memory model to be used, where *model* is one of:

t	tiny (cpu in non-segmented mode)
s	small (default)
m	medium
l	large
h	huge

Default:

-Ms

Description:

Select memory model to be used.

Example:

cp166 -Ml test.cc



Section *Memory Models* in chapter *Language Implementation* in the C166/ST10 C Cross-Compiler User's Manual.

--namespaces

Option:

--namespaces
--no-namespaces

Default:

--namespaces

Description:

Enable or disable support for namespaces.

Example:

```
cp166 --no-namespaces test.cc
```



--using-std

Section *Namespace Support* in chapter *Language Implementation*.

--new-for-init / --old-for-init

Option:

--new-for-init
--old-for-init

Default:

--new-for-init

Description:

Control the scope of a declaration in a **for-init-statement**. The old (cfront-compatible) scoping rules mean the declaration is in the scope to which the **for** statement itself belongs; the new (standard-conforming) rules in effect wrap the entire **for** statement in its own implicitly generated scope.

Example:

```
cp166 --old-for-init test.cc
```

--no-code-gen / -n

Option:

-n
--no-code-gen

Description:

Do syntax-checking only. Do not generate a C file.

Example:

```
cp166 --no-code-gen test.cc
```

--no-line-commands / -P

Option:

-P
--no-line-commands

Description:

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and without line control information. When you use the **-P** option, use the **--output** option to separate the output from the header produced by the compiler.

Example:

```
cp166 -P --output preout test.cc
```



--comments / -C, --preprocess / -E, --dependencies / -M

--nonconst-ref-anachronism

Option:

--nonconst-ref-anachronism
--no-nonconst-ref-anachronism

Default:

--nonconst-ref-anachronism

Description:

Enable or disable the anachronism of allowing a reference to nonconst to bind to a class rvalue of the right type. This anachronism is also enabled by the **--anachronisms** option and the cfront-compatibility options.

Example:

```
cp166  --no-nonconst-ref-anachronism test.cc
```



--anachronisms, --cfront-2.1 / -b / --cfront-3.0

Section *Anachronisms Accepted* in chapter *Language Implementation*.

--nonstd-qualifier-deduction

Option:

--nonstd-qualifier-deduction
--no-nonstd-qualifier-deduction

Default:

--no-nonstd-qualifier-deduction

Description:

Controls whether nonstandard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter **T** can be deduced in contexts like **A<T>::B** or **T::B**. The standard deduction mechanism treats these as nondeduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

Example:

```
cp166 --nonstd-qualifier-deduction test.cc
```

--nonstd-using-decl

Option:

--nonstd-using-decl
--no-nonstd-using-decl

Default:

--no-nonstd-using-decl

Description:

Controls whether a non-member using declaration that specifies an unqualified name is allowed.

Example:

```
cp166 --nonstd-using-decl test.cc
```

--no-preproc-only

Option:

--no-preproc-only

Description:

May be used in conjunction with the options that normally cause the C++ compiler to do preprocessing only (e.g., **--preprocess**, etc.) to specify that a full compilation should be done (not just preprocessing). When used with the implicit inclusion option, this makes it possible to generate a preprocessed output file that includes any implicitly included files.

Examples:

```
cp166 -E -B --no-preproc-only test.cc
```



--preprocess / -E,
--implicit-include / -B, --no-line-commands / -P

--no-use-before-set-warnings / -j

Option:

-j
--no-use-before-set-warnings

Description:

Suppress warnings on local automatic variables that are used before their values are set.

Example:

cp166 -j test.cc



--no-warnings / -w

--no-warnings / -w

Option:

-w
--no-warnings

Description:

Suppress all warning messages. Error messages are still issued.

Example:

To suppress all warnings, enter:

```
cp166 -w test.cc
```

--old-line-commands

Option:

--old-line-commands

Description:

When generating source output, put out `#line` directives in the form used by the Reiser cpp, that is, `# nnn` instead of `#line nnn`.

Example:

To do preprocessing only, without comments and with old style line control information, enter:

```
cp166 -E --old-line-commands test.cc
```



```
--preprocess / -E, --no-line-commands / -P
```

--old-specializations

Option:

--old-specializations
--no-old-specializations

Default:

--old-specializations

Description:

Enable or disable acceptance of old-style template specializations (that is, specializations that do not use the `template<>` syntax).

Example:

```
cp166 --no-old-specializations test.cc
```

--old-style-preprocessing

Option:

--old-style-preprocessing

Description:

Forces pcc style preprocessing when compiling. This may be used when compiling an ANSI C++ program on a system in which the system header files require pcc style preprocessing.

Example:

To force pcc style preprocessing, enter:

```
cp166 -E --old-style-preprocessing test.cc
```



```
--preprocess / -E, --no-line-commands / -P
```

--one-instantiation-per-object

Option:

--one-instantiation-per-object

Description:

Put out each template instantiation in this compilation (function or static data member) in a separate object file. The primary object file contains everything else in the compilation, that is, everything that is not an instantiation. Having each instantiation in a separate object file is very useful when creating libraries, because it allows the user of the library to pull in only the instantiations that are needed. That can be essential if two different libraries include some of the same instantiations.

Example:

To create separate instantiation files, enter:

```
cp166 --one-instantiation-per-object test.cc
```



Section *Template Instantiation* in chapter *Language Implementation*.

--output

Option:

--output *file*

Arguments:

An output filename specifying the preprocessing output file.

Default:

No preprocessing output file is generated.

Description:

Use *file* as output filename for the preprocessing output file.

Example:

To use the file `my.pre` as the preprocessing output file, enter:

```
cp166 -E --output my.pre test.cc
```



--preprocess / **-E**, **--no-line-commands** / **-P**

--pch

Option:

--pch

Description:

Automatically use and/or create a precompiled header file. For details, see the *Precompiled Headers* section in chapter *Language Implementation*.

If **--use-pch** or **--create-pch** (manual PCH mode) appears on the command line following this option, its effect is erased.

Example:

```
cp166 --pch test.cc
```



--use-pch, --create-pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--pch-dir

Option:

--pch-dir *dir_name*

Arguments:

The name of the directory to search for and/or create a precompiled header file.

Description:

Specify the directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (**--pch**) or manual PCH mode (**--create-pch** or **--use-pch**).

Example:

To use the directory `/usr/include/pch` to automatically create precompiled header files, enter:

```
cp166 --pch-dir /usr/include/pch --pch test.cc
```



--pch, --use-pch, --create-pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--pch-messages

Option:

--pch-messages
--no-pch-messages

Default:

--pch-messages

Description:

Enable or disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

Example:

```
cp166 --create-pch test.pch --pch-messages test.cc
```

```
"test.cc": creating precompiled header file "test.pch"
```



--pch, --use-pch, --create-pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--pch-verbose

Option:

--pch-verbose

Description:

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Example:

```
cp166 --pch --pch-verbose test.cc
```



--pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--pending-instantiations

Option:

--pending-instantiations *n*

Arguments:

The maximum number of instantiation for a single template.

Default:

64

Description:

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If *n* is zero, there is no limit.

Example:

To specify a maximum of 32 pending instantiations, enter:

```
cp166 --pending-instantiations 32 test.cc
```



Section *Template Instantiation* in chapter *Language Implementation*.

--preprocess / -E

Option:

-E
--preprocess

Description:

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and with line control information. When you use the **-E** option, use the **--output** option to separate the output from the header produced by the compiler.

Example:

```
cp166 -E --output preout test.cc
```



--comments / -C,
--dependencies / -M,
--no-line-commands / -P

--remarks / -r

Option:

-r
--remarks

Description:

Issue remarks, which are diagnostic messages even milder than warnings.

Example:

To enable the display of remarks, enter:

```
cp166 -r test.cc
```

--remove-unneeded-entities

Option:

--remove-unneeded-entities
--no-remove-unneeded-entities

Default:

--remove-unneeded-entities

Description:

Enable or disable an optimization to remove unneeded entities from the generated intermediate C file. Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.

Example:

```
cp166 --no-remove-unneeded-entities test.cc
```

--rtti

Option:

--rtti
--no-rtti

Default:

--no-rtti

Description:

Enable or disable support for RTTI (run-time type information) features: `dynamic_cast`, `typeid`.

Example:

```
cp166 --rtti test.cc
```

--signed-chars / -s

Option:

-s
--signed-chars

Description:

Treat 'character' type variables as 'signed character' variables. When plain char is signed, the macro `__SIGNED_CHARS__` is defined.

Example:

cp166 -s test.cc



--unsigned-chars / -u

--special-subscript-cost

Option:

--special-subscript-cost
--no-special-subscript-cost

Default:

--no-special-subscript-cost

Description:

Enable or disable a special nonstandard weighting of the conversion to the integral operand of the [] operator in overload resolution.

This is a compatibility feature that may be useful with some existing code. The special cost is enabled by default in cfront 3.0 mode. With this feature enabled, the following code compiles without error:

```
struct A {
    A();
    operator int *();
    int operator[](unsigned);
};
void main() {
    A a;
    a[0]; // Ambiguous, but allowed with this option
        // operator[] is chosen
}
```

Example:

cp166 --special-subscript-cost test.cc

--strict / -A

--strict-warnings / -a

Option:

-A / --strict

-a / --strict-warnings

Description:

Enable strict ANSI mode, which provides diagnostic messages when non-ANSI features are used, and disables features that conflict with ANSI C or C++. ANSI violations can be issued as either warnings or errors depending on which command line option is used. The **--strict** options issue errors and the **--strict-warnings** options issue warnings. The error threshold is set so that the requested diagnostics will be listed.

Example:

To enable strict ANSI mode, with error diagnostic messages, enter:

```
cp166 -A test.cc
```

--suppress-typeinfo-vars

Option:

--suppress-typeinfo-vars

Description:

Suppress the generation of type info variables when run-time type info (RTTI) is disabled. By default only type info variables are generated, no other run-time type info. With this option you can also suppress type info variables.

Example:

```
cp166 --suppress-typeinfo-vars test.cc
```



```
--rtti
```

--suppress-vtbl

Option:

--suppress-vtbl

Description:

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The **--suppress-vtbl** option suppresses the definition of the virtual function tables for such classes, and the **--force-vtbl** option forces the definition of the virtual function table for such classes. **--force-vtbl** differs from the default behavior in that it does not force the definition to be local.

Example:

```
cp166 --suppress-vtbl test.cc
```



```
--force-vtbl
```

--sys-include

Option:

--sys-include *directory*

Arguments:

The name of the system include directory to search for include file(s).

Description:

Change the algorithm for searching system include files whose names do not have an absolute pathname to look in *directory*.

Example:

```
cp166 --sys-include /proj/include test.cc
```



Section 3.2, *Include Files*.

--include-directory

--timing / -#

Option:

-#
--timing

Default:

No timing information is generated.

Description:

Generate compilation timing information. This option causes the compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

Example:

```
cp166  -# test.cc  
  
processed 180 lines at 8102 lines/min
```

--trace-includes / -H

Option:

-H
--trace-includes

Description:

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of the names of files #included.

Examples:

```
cp166 -H test.cc
```

```
iostream.h  
string.h
```



```
--preprocess / -E, --no-line-commands / -P
```

--tsw-diagnostics

Option:

--tsw-diagnostics
--no-tsw-diagnostics

Default:

--tsw-diagnostics

Description:

Enable or disable a mode in which the error message is given in the TASKING style. So, in the same format as the TASKING C compiler messages.

Example:

```
cp166  --no-tsw-diagnostics test.cc
```



--brief-diagnostics

Chapter *Compiler Diagnostics* and Appendix *Error Messages*.

--typename

Option:

--typename
--no-typename

Default:

--typename

Description:

Enable or disable recognition of the `typename` keyword.

Example:

```
cp166 --no-typename test.cc
```



--implicit-typename

--undefine / -U

Option:

-U*name*
--undefine *name*

Arguments:

The name macro you want to undefine.

Description:

Remove any initial definition of identifier *name* as in `#undef`, unless it is a predefined ANSI standard macro. ANSI specifies the following predefined symbols to exist, which cannot be removed:

<code>__FILE__</code>	"current source filename"
<code>__LINE__</code>	current source line number (int type)
<code>__TIME__</code>	"hh:mm:ss"
<code>__DATE__</code>	"Mmm dd yyyy"
<code>__STDC__</code>	level of ANSI standard. This macro is set to 1 when the option to disable language extensions (-A) is effective. Whenever language extensions are excepted, <code>__STDC__</code> is set to 0 (zero).

`__cplusplus` is defined when compiling a C++ program

When **cp166** is invoked, also the following predefined symbols exist:

`c_plusplus` is defined in addition to the standard `__cplusplus`

`__SIGNED_CHARS__`
 is defined when plain **char** is signed.

`__WCHAR_T` is defined when **wchar_t** is a keyword.

`__BOOL` is defined when **bool** is a keyword.

`__ARRAY_OPERATORS`
 is defined when array new and delete are enabled.

These symbols can be turned off with the **-U** option.

Example:

```
cp166 -Uc_plusplus test.cc
```



-D / --define

--unsigned-chars / -u

Option:

-u
--unsigned-chars

Description:

Treat 'character' type variables as 'unsigned character' variables.

Example:

cp166 -u test.cc



--signed-chars / -s

--use-pch

Option:

--use-pch *filename*

Arguments:

The filename to use as a precompiled header file.

Description:

Use a precompiled header file of the specified name as part of the current compilation. If **--pch** (automatic PCH mode) or **--create-pch** appears on the command line following this option, its effect is erased.

Example:

To use the precompiled header file with the name `test.pch`, enter:

```
cp166 --use-pch test.pch test.cc
```



--pch, --create-pch

Section *Precompiled Headers* in chapter *Language Implementation*.

--using-std

Option:

--using-std
--no-using-std

Default:

--using-std

Description:

Enable or disable implicit use of the `std` namespace when standard header files are included.

Example:

```
cp166 --using-std test.cc
```



--namespaces

Section *Namespace Support* in chapter *Language Implementation*.

--variadic-macros

Option:

--variadic-macros
--no-variadic-macros

Default:

--no-variadic-macros

Description:

Allow or disallow macros with a variable number of arguments.

Example:

```
cp166 --variadic-macros test.cc
```



```
--extended-variadic-macros
```

--version / -V / -v

Option:

-V
-v
--version

Description:

Display version information.

Example:

cp166 -V

```
TASKING C166/ST10 C++ compiler    vx.yrz Build nnn  
Copyright years Altium BV        Serial# 00000000
```


--wchar_t-keyword

Option:

--wchar_t-keyword
--no-wchar_t-keyword

Default:

--wchar_t-keyword

Description:

Enable or disable recognition of `wchar_t` as a keyword.

Example:

```
cp166 --no-wchar_t-keyword test.cc
```

--wrap-diagnostics

Option:

--wrap-diagnostics
--no-wrap-diagnostics

Default:

--wrap-diagnostics

Description:

Enable or disable a mode in which the error message text is not wrapped when too long to fit on a single line.

Example:

```
cp166 --no-wrap-diagnostics test.cc
```



--brief-diagnostics

Chapter *Compiler Diagnostics* and Appendix *Error Messages*.

--xref / -X

Option:

-X*xfile*
--xref *xfile*

Arguments:

The name of the cross-reference file.

Description:

Generate cross-reference information in the file *xfile*. For each reference to an identifier in the source program, a line of the form

symbol_id name X file-name line-number column-number

is written, where *X* is

- D** for definition;
- d** for declaration (that is, a declaration that is not a definition);
- M** for modification;
- A** for address taken;
- U** for used;
- C** for changed (but actually meaning used and modified in a single operation, such as an increment);
- R** for any other kind of reference, or
- E** for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

3.2 INCLUDE FILES

You may specify include files in two ways: enclosed in `<...>` or enclosed in `"..."`. When an `#include` directive is seen, the following algorithm is used to try to open the include file:

1. If the filename is enclosed in `"..."`, and it is not an absolute pathname (does not begin with a `'\'` for PC, or a `'/'` for UNIX), the include file is searched for in the directory of the file containing the `#include` line. For example, in:

PC:

```
cp166 ..\..\source\test.cc
```

UNIX:

```
cp166 ../../source/test.cc
```

cp166 first searches in the directory `..\..\source` (`../../source` for UNIX) for include files.

If you compile a source file in the directory where the file is located (**cp166 test.cc**), the compiler searches for include files in the current directory.



This first step is not done for include files enclosed in `<...>`.

2. Use the directories specified with the **-I** or **--include-directory** option, in a left-to-right order. For example:

PC:

```
cp166 -I..\..\include demo.cc
```

UNIX:

```
cp166 -I../../include demo.cc
```

3. Check if the environment variable `CP166INC` exists. If it does exist, use the contents as a directory specifier for include files. You can specify more than one directory in the environment variable `CP166INC` by using a separator character. Instead of using **-I** as in the example above, you can specify the same directory using `CP166INC`:

PC:

```
set CP166INC=../../include
cp166 demo.cc
```

UNIX:

if using the Bourne shell (sh)

```
CP166INC=../../include
export CP166INC
cp166 demo.cc
```

or if using the C-shell (csh)

```
setenv CP166INC ../../include
cp166 demo.cc
```

4. When an include file is not found with the rules mentioned above, the compiler tries the subdirectories **include.cpp** and **include**, one directory higher than the directory containing the **cp166** binary. For example:

PC:

cp166.exe is installed in the directory **C:\C166\BIN**
 The directories searched for the include file are
C:\C166\INCLUDE.CPP and **C:\C166\INCLUDE**

UNIX:

cp166 is installed in the directory **/usr/local/c166/bin**
 The directories searched for the include file are
/usr/local/c166/include.cpp and
/usr/local/c166/include

The compiler determines run-time which directory the binary is executed from to find this **include** directory.

5. If the include file is still not found, the directories specified in the **--sys-include** option are searched.

A directory name specified with the **-I** option or in **CP166INC** may or may not be terminated with a directory separator, because **cp166** inserts this separator, if omitted.

When you specify more than one directory to the environment variable CP166INC, you have to use one of the following separator characters:

PC:

`;` , *space*

e.g. **set CP166INC=..\..\include;\proj\include**

UNIX:

`:` ; , *space*

e.g. **setenv CP166INC ../../include:/proj/include**

If the include directory is specified as `-`, e.g., **-I-**, the option indicates the point in the list of **-I** or **--include-directory** options at which the search for file names enclosed in `<...>` should begin. That is, the search for `<...>` names should only consider directories named in **-I** or **--include-directory** options following the **-I-**, and the directories of items 3 and 4 above. **-I-** also removes the directory containing the current input file (item 1 above) from the search path for file names enclosed in `"..."`.

An include directory specified with the **--sys-include** option is considered a “system” include directory. Warnings are suppressed when processing files found in system include directories.

If the filename has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the next search directory. The default set of suffixes is, no extension, `.h` and `.hpp`. The default can be overridden using the **--incl-suffixes** command line option. A null file suffix cannot be used unless it is present in the suffix list (that is, the C++ compiler will always attempt to add a suffix from the suffix list when the filename has no suffix).

3.3 PRAGMAS

According to ANSI (3.8.6) a preprocessing directive of the form:

```
#pragma pragma-token-list new-line
```

causes the compiler to behave in an implementation-defined manner. The compiler ignores pragmas which are not mentioned in the list below. Pragmas give directions to the code generator of the compiler. Besides the pragmas there are two other possibilities to steer the code generator: command line options and keywords. The compiler acknowledges these three groups using the following rule:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. So the pragma has the highest priority.

This approach makes it possible to set a default optimization level for a source module, which can be overridden temporarily within the source by a pragma.

cp166 supports the following pragmas and all pragmas that are described in the *C Cross-Compiler User's Manual*:

instantiate

do_not_instantiate

can_instantiate

These are template instantiation pragmas. They are described in detail in the section *Template Instantiation* in chapter *Language Implementation*.

hdrstop

no_pch

These are precompiled header pragmas. They are described in detail in the section *Precompiled Headers* in chapter *Language Implementation*.

once

When placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the C++ compiler sees **#pragma once** at the start of a header file, it will skip over it if the file is *#included* again.

A typical idiom is to place an *#ifndef* guard around the body of the file, with a *#define* of the guard variable after the *#ifndef*:

```
#pragma once    // optional
#ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
```

The **#pragma once** is marked as optional in this example, because the C++ compiler recognizes the `#ifndef` idiom and does the optimization even in its absence. **#pragma once** is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

ident

This pragma is given in the form:

```
#pragma ident "string"
```

or:

```
#ident "string"
```


3.4 COMPILER LIMITS

The ANSI C standard [1-2.2.4] defines a number of translation limits, which a C compiler must support to conform to the standard. The standard states that a compiler implementation should be able to translate and execute a program that contains at least one instance of every one of the limits listed below. The C compiler's actual limits are given within parentheses.

Most of the actual compiler limits are determined by the amount of free memory in the host system. In this case a 'D' (Dynamic) is given between parentheses. Some limits are determined by the size of the internal compiler parser stack. These limits are marked with a 'P'. Although the size of this stack is 200, the actual limit can be lower and depends on the structure of the translated program.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (P > 15)
- 8 nesting levels of conditional inclusion (50)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration (15)
- 31 nesting levels of parenthesized declarators within a full declarator (P > 31)
- 32 nesting levels of parenthesized expressions within a full expression (P > 32)
- 31 significant characters in an external identifier (full ANSI-C mode),
120 significant characters in an external identifier (non ANSI-C mode)
- 511 external identifiers in one translation unit (D)
- 127 identifiers with block scope declared in one block (D)
- 1024 macro identifiers simultaneously defined in one translation unit (D)
- 31 parameters in one function declaration (D)
- 31 arguments in one function call (D)
- 31 parameters in one macro definition (D)
- 31 arguments in one macro call (D)
- 509 characters in a logical source line (1500)
- 509 characters in a character string literal or wide string literal (after concatenation) (1500)

- 8 nesting levels for **#included** files (50)
- 257 case labels for a switch statement, excluding those for any nested switch statements (D)
- 127 members in a single structure or union (D)
- 127 enumeration constants in a single enumeration (D)
- 15 levels of nested structure or union definitions in a single struct-declaration-list (D)



CHAPTER

4

COMPILER DIAGNOSTICS



4

CHAPTER

4.1 DIAGNOSTIC MESSAGES

Diagnostic messages have an associated *severity*, as follows:

- Catastrophic errors, also called 'fatal errors', indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- Errors indicate violations of the syntax or semantic rules of the C++ language. Compilation continues, but object code is not generated.
- Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected).
- The last class of messages are the internal compiler errors. These errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to TASKING, using a Problem Report form. Please include a diskette or tape, containing a small C++ program causing the error.

By default, **--tsw-diagnostics**, diagnostics are written to `stderr` with a form like the following:

```
test.cc
    5:  break;
E 116: a break statement may only be used within a loop or switch
```

With the command line option **--no-tsw-diagnostics** the message appear in the following form:

```
"test.cc", line 5: a break statement may only be used within a loop
                  or switch
                  break;
                  ^
```



Note that the message identifies the file and line involved, and that the source line itself (with position indicated by the ^) follows the message. If there are several diagnostics in one source line, each diagnostic will have the form above, with the result that the text of the source line will be displayed several times, with an appropriate position each time.

Long messages are wrapped to additional lines when necessary.

A configuration flag controls whether or not the string `error:` appears, i.e., the C++ compiler can be configured so that the severity string is omitted when the severity is `error`.

The command line option **--brief-diagnostics** may be used to request a shorter form of the diagnostic output in which the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

The command line option **--display-error-number** may be used to request that the error number be included in the diagnostic message. When displayed, the error number also indicates whether the error may have its severity overridden on the command line (with one of the **--diag-severity** options). If the severity may be overridden, the error number will include the suffix **-D** (for discretionary); otherwise no suffix will be present.

```
"Test_name.cc", line 7: error #64-D: declaration does not
    declare anything
    struct ;
    ^
```

```
"Test_name.cc", line 9: error #77: this declaration has no storage
    class or type specifier
    xxxxxx;
    ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, a given error may be discretionary in some cases and not in others.

For some messages, a list of entities is useful; they are listed following the initial error message:

```
"test.cc", line 4: error: more than one instance of overloaded
    function "f" matches the argument list:
        function "f(int)"
        function "f(float)"
        argument types are: (double)
    f(1.5);
    ^
```

In some cases, some additional context information is provided; specifically, such context information is useful when the C++ compiler issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
"test.cc", line 7: error: "A::A()" is inaccessible
  B x;
    ^
```

detected during implicit generation of "B::B()" at line 7

Without the context information, it is very hard to figure out what the error refers to.



For a list of error messages and error numbers, see Appendix A, *Error Messages*.

4.2 TERMINATION MESSAGES

cp166 writes sign-off messages to **stderr** if errors are detected. For example, one of the following forms of message

```
n errors detected in the compilation of "ifile".
```

```
1 catastrophic error detected in the compilation of "ifile".
```

```
n errors and 1 catastrophic error detected in the compilation of
"ifile".
```

is written to indicate the detection of errors in the compilation. No message is written if no errors were detected. The following message

Error limit reached.

is written when the count of errors reaches the error limit (see the **-e** option); compilation is then terminated. The message

Compilation terminated.

is written at the end of a compilation that was prematurely terminated because of a catastrophic error. The message

Compilation aborted

is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler. If such an internal error appears, please report the occurrence to TASKING, using a Problem Report form. Please include a diskette or tape, containing a small C++ program causing the error.

4.3 RESPONSE TO SIGNALS

The signals **SIGINT** (caused by a user interrupt, like **^C**) and **SIGTERM** (caused by a **kill** command) are trapped by the C++ compiler and cause abnormal termination.

4.4 RETURN VALUES

cp166 returns an exit status to the operating system environment for testing.

For example,

in a PC BATCH-file you can examine the exit status of the program executed with **ERRORLEVEL**:

```
cp166 %1.cc
IF ERRORLEVEL 1 GOTO STOP_BATCH
```

In a Bourne shell script, the exit status can be found in the **\$?** variable, for example:

```
cp166 $*
case $? in
0)          echo ok ;;
2|4)       echo error ;;
esac
```

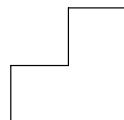
The exit status of **cp166** indicates the highest severity diagnostic detected and is one of the numbers of the following list:

- 1 Abnormal termination
- 0 Compilation successful, no errors, maybe some remarks
- 0 There were warnings
- 2 There were user errors, but terminated normally
- 4 A catastrophic error, premature ending

APPENDIX

ERROR MESSAGES

A



A

APPENDIX

1 INTRODUCTION

This appendix lists all diagnostic messages, starting with the error number and the error tag name, followed by the message itself. The error number and/or error tag can be used in **--diag-severity** options to override the normal error severity.

The C++ compiler produces error messages on standard error output. With the **--error-output** option you can redirect the error messages to an error list file.

Normally, diagnostics are written to `stderr` in the following form (TASKING layout):

severity #err_num: message

The *severity* can be one of: **R** (remark), **W** (warning), **E** (error), **F** (fatal error), **S** (internal error).

With **--no-tsw-diagnostics**, diagnostics are written to `stderr` in the following form:

"filename", line line_num: message

With **--display-error-number** this form will be:

"filename", line line_num: severity #err_num: message

or:

"filename", line line_num: severity #err_num-D: message

Where *severity* can be one of: remark, warning, error, catastrophic error, command-line error or internal error.

If the severity may be overridden, the error number will include the suffix **-D** (for discretionary); otherwise no suffix will be present.

In a raw listing file (**-L** option) diagnostic messages have the following layout, starting with the severity (R: remark, W: warning, E: error, C: catastrophe):

[R | W | E | C] "filename" line_number column_number error_message



For more detailed information see chapter *Compiler Diagnostics*.

All diagnostic messages are listed below.

2 MESSAGES

- 0001 last_line_incomplete:
last line of file ends without a newline
- 0002 last_line_backslash:
last line of file ends with a backslash
- 0003 include_recursion:
#include file "xxxx" includes itself
- 0004 out_of_memory:
out of memory
- 0005 source_file_could_not_be_opened:
could not open source file "xxxx"
- 0006 comment_unclosed_at_eof:
comment unclosed at end of file
- 0007 bad_token:
unrecognized token
- 0008 unclosed_string:
missing closing quote
- 0009 nested_comment:
nested comment is not allowed
- 0010 bad_use_of_sharp:
"#" not expected here
- 0011 bad_pp_directive_keyword:
unrecognized preprocessing directive
- 0012 end_of_flush:
parsing restarts here after previous syntax error
- 0013 exp_file_name:
expected a file name

- 0014 `extra_text_in_pp_directive:`
extra text after expected end of preprocessing directive
- 0016 `illegal_source_file_name:`
"xxxx" is not a valid source file name
- 0017 `exp_rbracket:`
expected a "]"
- 0018 `exp_rparen:`
expected a ")"
- 0019 `extra_chars_on_number:`
extra text after expected end of number
- 0020 `undefined_identifier:`
identifier "xxxx" is undefined
- 0021 `useless_type_qualifiers:`
type qualifiers are meaningless in this declaration
- 0022 `bad_hex_digit:`
invalid hexadecimal number
- 0023 `integer_too_large:`
integer constant is too large
- 0024 `bad_octal_digit:`
invalid octal digit
- 0025 `zero_length_string:`
quoted string should contain at least one character
- 0026 `too_many_characters:`
too many characters in character constant
- 0027 `bad_character_value:`
character value is out of range
- 0028 `expr_not_constant:`
expression must have a constant value

- 0029 `exp_primary_expr`:
expected an expression
- 0030 `bad_float_value`:
floating constant is out of range
- 0031 `expr_not_integral`:
expression must have integral type
- 0032 `expr_not_arithmetic`:
expression must have arithmetic type
- 0033 `exp_line_number`:
expected a line number
- 0034 `bad_line_number`:
invalid line number
- 0035 `error_directive`:
#error directive: *xxxx*
- 0036 `missing_pp_if`:
the #if for this directive is missing
- 0037 `missing_endif`:
the #endif for this directive is missing
- 0038 `pp_else_already_appeared`:
directive is not allowed — an #else has already appeared
- 0039 `divide_by_zero`:
division by zero
- 0040 `exp_identifier`:
expected an identifier
- 0041 `expr_not_scalar`:
expression must have arithmetic or pointer type
- 0042 `incompatible_operands`:
operand types are incompatible ("*type*" and "*type*")

- 0044 `expr_not_pointer:`
expression must have pointer type
- 0045 `cannot_undef_predef_macro:`
`#undef` may not be used on this predefined name
- 0046 `cannot_redef_predef_macro:`
this predefined name may not be redefined
- 0047 `bad_macro_redef:`
incompatible redefinition of macro *"entity"* (declared at line *xxxx*)
- 0049 `duplicate_macro_param_name:`
duplicate macro parameter name
- 0050 `paste_cannot_be_first:`
`"##"` may not be first in a macro definition
- 0051 `paste_cannot_be_last:`
`"##"` may not be last in a macro definition
- 0052 `exp_macro_param:`
expected a macro parameter name
- 0053 `exp_colon:`
expected a `":"`
- 0054 `too_few_macro_args:`
too few arguments in macro invocation
- 0055 `too_many_macro_args:`
too many arguments in macro invocation
- 0056 `sizeof_function:`
operand of `sizeof` may not be a function
- 0057 `bad_constant_operator:`
this operator is not allowed in a constant expression
- 0058 `bad_pp_operator:`
this operator is not allowed in a preprocessing expression

- 0059 `bad_constant_function_call`:
function call is not allowed in a constant expression
- 0060 `bad_integral_operator`:
this operator is not allowed in an integral constant expression
- 0061 `integer_overflow`:
integer operation result is out of range
- 0062 `negative_shift_count`:
shift count is negative
- 0063 `shift_count_too_large`:
shift count is too large
- 0064 `useless_decl`:
declaration does not declare anything
- 0065 `exp_semicolon`:
expected a ";"
- 0066 `enum_value_out_of_int_range`:
enumeration value is out of "int" range
- 0067 `exp_rbrace`:
expected a "}"
- 0068 `integer_sign_change`:
integer conversion resulted in a change of sign
- 0069 `integer_truncated`:
integer conversion resulted in truncation
- 0070 `incomplete_type_not_allowed`:
incomplete type is not allowed
- 0071 `sizeof_bit_field`:
operand of sizeof may not be a bit field
- 0075 `bad_indirection_operand`:
operand of "*" must be a pointer

- 0076 `empty_macro_argument`:
argument to macro is empty
- 0077 `missing_decl_specifiers`:
this declaration has no storage class or type specifier
- 0078 `initializer_in_param`:
a parameter declaration may not have an initializer
- 0079 `exp_type_specifier`:
expected a type specifier
- 0080 `storage_class_not_allowed`:
a storage class may not be specified here
- 0081 `mult_storage_classes`:
more than one storage class may not be specified
- 0082 `storage_class_not_first`:
storage class is not first
- 0083 `dupl_type_qualifier`:
type qualifier specified more than once
- 0084 `bad_combination_of_type_specifiers`:
invalid combination of type specifiers
- 0085 `bad_param_storage_class`:
invalid storage class for a parameter
- 0086 `bad_function_storage_class`:
invalid storage class for a function
- 0087 `type_specifier_not_allowed`:
a type specifier may not be used here
- 0088 `array_of_function`:
array of functions is not allowed
- 0089 `array_of_void`:
array of void is not allowed

- 0090 `function_returning_function`:
function returning function is not allowed
- 0091 `function_returning_array`:
function returning array is not allowed
- 0092 `param_id_list_needs_function_def`:
identifier-list parameters may only be used in a function definition
- 0093 `function_type_must_come_from_declarator`:
function type may not come from a typedef
- 0094 `array_size_must_be_positive`:
the size of an array must be greater than zero
- 0095 `array_size_too_large`:
array is too large
- 0096 `empty_translation_unit`:
a translation unit must contain at least one declaration
- 0097 `bad_function_return_type`:
a function may not return a value of this type
- 0098 `bad_array_element_type`:
an array may not have elements of this type
- 0099 `decl_should_be_of_param`:
a declaration here must declare a parameter
- 0100 `dupl_param_name`:
duplicate parameter name
- 0101 `id_already_declared`:
"xxxx" has already been declared in the current scope
- 0102 `nonstd_forward_decl_enum`:
forward declaration of enum type is nonstandard
- 0103 `class_too_large`:
class is too large

- 0104 `struct_too_large`:
struct or union is too large
- 0105 `bad_bit_field_size`:
invalid size for bit field
- 0106 `bad_bit_field_type`:
invalid type for a bit field
- 0107 `zero_length_bit_field_must_be_unnamed`:
zero-length bit field must be unnamed
- 0108 `signed_one_bit_field`:
signed bit field of length 1
- 0109 `expr_not_ptr_to_function`:
expression must have (pointer-to-) function type
- 0110 `exp_definition_of_tag`:
expected either a definition or a tag name
- 0111 `code_is_unreachable`:
statement is unreachable
- 0112 `exp_while`:
expected "while"
- 0114 `never_defined`:
entity-kind "entity" was referenced but not defined
- 0115 `continue_must_be_in_loop`:
a continue statement may only be used within a loop
- 0116 `break_must_be_in_loop_or_switch`:
a break statement may only be used within a loop or switch
- 0117 `no_value_returned_in_non_void_function`:
non-void *entity-kind "entity"* (declared at line `xxxx`) should return a value

- 0118 `value_returned_in_void_function`:
a void function may not return a value
- 0119 `cast_to_bad_type`:
cast to type "*type*" is not allowed
- 0120 `bad_return_value_type`:
return value type does not match the function type
- 0121 `case_label_must_be_in_switch`:
a case label may only be used within a switch
- 0122 `default_label_must_be_in_switch`:
a default label may only be used within a switch
- 0123 `case_label_appears_more_than_once`:
case label value has already appeared in this switch
- 0124 `default_label_appears_more_than_once`:
default label has already appeared in this switch
- 0125 `exp_lparen`:
expected a "("
- 0126 `expr_not_an_lvalue`:
expression must be an lvalue
- 0127 `exp_statement`:
expected a statement
- 0128 `loop_not_reachable`:
loop is not reachable from preceding code
- 0129 `block_scope_function_must_be_extern`:
a block-scope function may only have extern storage class
- 0130 `exp_lbrace`:
expected a "{"
- 0131 `expr_not_ptr_to_class`:
expression must have pointer-to-class type

- 0132 `expr_not_ptr_to_struct_or_union`:
expression must have pointer-to-struct-or-union type
- 0133 `exp_member_name`:
expected a member name
- 0134 `exp_field_name`:
expected a field name
- 0135 `not_a_member`:
entity-kind "entity" has no member "xxxx"
- 0136 `not_a_field`:
entity-kind "entity" has no field "xxxx"
- 0137 `expr_not_a_modifiable_lvalue`:
expression must be a modifiable lvalue
- 0138 `address_of_register_variable`:
taking the address of a register variable is not allowed
- 0139 `address_of_bit_field`:
taking the address of a bit field is not allowed
- 0140 `too_many_arguments`:
too many arguments in function call
- 0141 `all_proto_params_must_be_named`:
unnamed prototyped parameters not allowed when body is present
- 0142 `expr_not_pointer_to_object`:
expression must have pointer-to-object type
- 0143 `program_too_large`:
program too large or complicated to compile
- 0144 `bad_initializer_type`:
a value of type "*type*" cannot be used to initialize an entity of type "*type*"

- 0145 cannot_initialize:
entity-kind "entity" may not be initialized
- 0146 too_many_initializer_values:
too many initializer values
- 0147 not_compatible_with_previous_decl:
declaration is incompatible with *entity-kind "entity"* (declared at line xxxx)
- 0148 already_initialized:
entity-kind "entity" has already been initialized
- 0149 bad_file_scope_storage_class:
a global-scope declaration may not have this storage class
- 0150 type_cannot_be_param_name:
a type name may not be redeclared as a parameter
- 0151 typedef_cannot_be_param_name:
a typedef name may not be redeclared as a parameter
- 0152 non_zero_int_conv_to_pointer:
conversion of nonzero integer to pointer
- 0153 expr_not_class:
expression must have class type
- 0154 expr_not_struct_or_union:
expression must have struct or union type
- 0155 old_fashioned_assignment_operator:
old-fashioned assignment operator
- 0156 old_fashioned_initializer:
old-fashioned initializer
- 0157 expr_not_integral_constant:
expression must be an integral constant expression

- 0158 `expr_not_an_lvalue_or_function_designator`:
expression must be an lvalue or a function designator
- 0159 `decl_incompatible_with_previous_use`:
declaration is incompatible with previous "*entity*" (declared at line *xxxx*)
- 0160 `external_name_clash`:
name conflicts with previously used external name "*xxxx*"
- 0161 `unrecognized_pragma`:
unrecognized `#pragma`
- 0163 `cannot_open_temp_file`:
could not open temporary file "*xxxx*"
- 0164 `temp_file_dir_name_too_long`:
name of directory for temporary files is too long ("*xxxx*")
- 0165 `too_few_arguments`:
too few arguments in function call
- 0166 `bad_float_constant`:
invalid floating constant
- 0167 `incompatible_param`:
argument of type "*type*" is incompatible with parameter of type "*type*"
- 0168 `function_type_not_allowed`:
a function type is not allowed here
- 0169 `exp_declaration`:
expected a declaration
- 0170 `pointer_outside_base_object`:
pointer points outside of underlying object
- 0171 `bad_cast`:
invalid type conversion

- 0172 `linkage_conflict`:
external/internal linkage conflict with previous declaration
- 0173 `float_to_integer_conversion`:
floating-point value does not fit in required integral type
- 0174 `expr_has_no_effect`:
expression has no effect
- 0175 `subscript_out_of_range`:
subscript out of range
- 0177 `declared_but_not_referenced`:
entity-kind "entity" was declared but never referenced
- 0178 `pcc_address_of_array`:
"&" applied to an array has no effect
- 0179 `mod_by_zero`:
right operand of "%" is zero
- 0180 `old_style_incompatible_param`:
argument is incompatible with formal parameter
- 0181 `printf_arg_mismatch`:
argument is incompatible with corresponding format string conversion
- 0182 `empty_include_search_path`:
could not open source file "xxxx" (no directories in search list)
- 0183 `cast_not_integral`:
type of cast must be integral
- 0184 `cast_not_scalar`:
type of cast must be arithmetic or pointer
- 0185 `initialization_not_reachable`:
dynamic initialization in unreachable code

- 0186 unsigned_compare_with_zero:
pointless comparison of unsigned integer with zero
- 0187 assign_where_compare_meant:
use of "=" where "==" may have been intended
- 0188 mixed_enum_type:
enumerated type mixed with another type
- 0189 file_write_error:
error while writing xxxx file
- 0190 bad_il_file:
invalid intermediate language file
- 0191 cast_to_qualified_type:
type qualifier is meaningless on cast type
- 0192 unrecognized_char_escape:
unrecognized character escape sequence
- 0193 undefined_preproc_id:
zero used for undefined preprocessing identifier
- 0194 exp_asm_string:
expected an asm string
- 0195 asm_func_must_be_prototyped:
an asm function must be prototyped
- 0196 bad_asm_func_ellipsis:
an asm function may not have an ellipsis
- 0219 file_delete_error:
error while deleting file "xxxx"
- 0220 integer_to_float_conversion:
integral value does not fit in required floating-point type
- 0221 float_to_float_conversion:
floating-point value does not fit in required floating-point type

- 0222 `bad_float_operation_result`:
floating-point operation result is out of range
- 0223 `implicit_func_decl`:
function declared implicitly
- 0224 `too_few_printf_args`:
the format string requires additional arguments
- 0225 `too_many_printf_args`:
the format string ends before this argument
- 0226 `bad_printf_format_string`:
invalid format string conversion
- 0227 `macro_recursion`:
macro recursion
- 0228 `nonstd_extra_comma`:
trailing comma is nonstandard
- 0229 `enum_bit_field_too_small`:
bit field cannot contain all values of the enumerated type
- 0230 `nonstd_bit_field_type`:
nonstandard type for a bit field
- 0231 `decl_in_prototype_scope`:
declaration is not visible outside of function
- 0232 `decl_of_void_ignored`:
old-fashioned typedef of "void" ignored
- 0233 `old_fashioned_field_selection`:
left operand is not a struct or union containing this field
- 0234 `old_fashioned_ptr_field_selection`:
pointer does not point to struct or union containing this field
- 0235 `var_retained_incomp_type`:
variable "xxxx" was declared with a never-completed type

- 0236 `boolean_controlling_expr_is_constant`:
controlling expression is constant
- 0237 `switch_selector_expr_is_constant`:
selector expression is constant
- 0238 `bad_param_specifier`:
invalid specifier on a parameter
- 0239 `bad_specifier_outside_class_decl`:
invalid specifier outside a class declaration
- 0240 `dupl_decl_specifier`:
duplicate specifier in declaration
- 0241 `base_class_not_allowed_for_union`:
a union is not allowed to have a base class
- 0242 `access_already_specified`:
multiple access control specifiers are not allowed
- 0243 `missing_class_definition`:
class or struct definition is missing
- 0244 `name_not_member_of_class_or_base_classes`:
qualified name is not a member of class "*type*" or its base classes
- 0245 `member_ref_requires_object`:
a nonstatic member reference must be relative to a specific object
- 0246 `nonstatic_member_def_not_allowed`:
a nonstatic data member may not be defined outside its class
- 0247 `already_defined`:
entity-kind "*entity*" has already been defined
- 0248 `pointer_to_reference`:
pointer to reference is not allowed
- 0249 `reference_to_reference`:
reference to reference is not allowed

- 0250 `reference_to_void`:
reference to void is not allowed
- 0251 `array_of_reference`:
array of reference is not allowed
- 0252 `missing_initializer_on_reference`:
reference *entity-kind* "entity" requires an initializer
- 0253 `exp_comma`:
expected a ","
- 0254 `type_identifier_not_allowed`:
type name is not allowed
- 0255 `type_definition_not_allowed`:
type definition is not allowed
- 0256 `bad_type_name_redeclaration`:
invalid redeclaration of type name "entity" (declared at line .xxxx)
- 0257 `missing_initializer_on_const`:
const *entity-kind* "entity" requires an initializer
- 0258 `this_used_incorrectly`:
"this" may only be used inside a nonstatic member function
- 0259 `constant_value_not_known`:
constant value is not known
- 0260 `missing_type_specifier`:
explicit type is missing ("int" assumed)
- 0261 `missing_access_specifier`:
access control not specified ("xxxx" by default)
- 0262 `not_a_class_or_struct_name`:
not a class or struct name
- 0263 `dupl_base_class_name`:
duplicate base class name

- 0264 `bad_base_class`:
invalid base class
- 0265 `no_access_to_name`:
entity-kind "*entity*" is inaccessible
- 0266 `ambiguous_name`:
"*entity*" is ambiguous
- 0267 `old_style_parameter_list`:
old-style parameter list (anachronism)
- 0268 `declaration_after_statements`:
declaration may not appear after executable statement in block
- 0269 `inaccessible_base_class`:
implicit conversion to inaccessible base class "*type*" is not allowed
- 0274 `improperly_terminated_macro_call`:
improperly terminated macro invocation
- 0276 `id_must_be_class_or_namespace_name`:
name followed by "::<" must be a class or namespace name
- 0277 `bad_friend_decl`:
invalid friend declaration
- 0278 `value_returned_in_constructor`:
a constructor or destructor may not return a value
- 0279 `bad_destructor_decl`:
invalid destructor declaration
- 0280 `class_and_member_name_conflict`:
invalid declaration of a member with the same name as its class
- 0281 `global_qualifier_not_allowed`:
global-scope qualifier (leading "::<") is not allowed
- 0282 `name_not_found_in_file_scope`:
the global scope has no "xxxx"

- 0283 `qualified_name_not_allowed`:
qualified name is not allowed
- 0284 `null_reference`:
NULL reference is not allowed
- 0285 `brace_initialization_not_allowed`:
initialization with "{...}" is not allowed for object of type "*type*"
- 0286 `ambiguous_base_class`:
base class "*type*" is ambiguous
- 0287 `ambiguous_derived_class`:
derived class "*type*" contains more than one instance of class "*type*"
- 0288 `derived_class_from_virtual_base`:
cannot convert pointer to base class "*type*" to pointer to derived class "*type*" — base class is virtual
- 0289 `no_matching_constructor`:
no instance of constructor "*entity*" matches the argument list
- 0290 `ambiguous_copy_constructor`:
copy constructor for class "*type*" is ambiguous
- 0291 `no_default_constructor`:
no default constructor exists for class "*type*"
- 0292 `not_a_field_or_base_class`:
"*xxxx*" is not a nonstatic data member or base class of class "*type*"
- 0293 `indirect_nonvirtual_base_class_not_allowed`:
indirect nonvirtual base class is not allowed
- 0294 `bad_union_field`:
invalid union member — class "*type*" has a disallowed member function
- 0296 `bad_rvalue_array`:
invalid use of non-lvalue array

- 0297 `exp_operator`:
expected an operator
- 0298 `inherited_member_not_allowed`:
inherited member is not allowed
- 0299 `indeterminate_overloaded_function`:
cannot determine which instance of *entity-kind* "*entity*" is intended
- 0300 `bound_function_must_be_called`:
a pointer to a bound function may only be used to call the function
- 0301 `duplicate_typedef`:
typedef name has already been declared (with same type)
- 0302 `function_redefinition`:
entity-kind "*entity*" has already been defined
- 0304 `no_matching_function`:
no instance of *entity-kind* "*entity*" matches the argument list
- 0305 `type_def_not_allowed_in_func_type_decl`:
type definition is not allowed in function return type declaration
- 0306 `default_arg_not_at_end`:
default argument not at end of parameter list
- 0307 `default_arg_already_defined`:
redefinition of default argument
- 0308 `ambiguous_overloaded_function`:
more than one instance of *entity-kind* "*entity*" matches the argument list:
- 0309 `ambiguous_constructor`:
more than one instance of constructor "*entity*" matches the argument list:
- 0310 `bad_default_arg_type`:
default argument of type "*type*" is incompatible with parameter of type "*type*"

- 0311 `return_type_cannot_distinguish_functions`:
cannot overload functions distinguished by return type alone
- 0312 `no_user_defined_conversion`:
no suitable user-defined conversion from "*type*" to "*type*" exists
- 0313 `function_qualifier_not_allowed`:
type qualifier is not allowed on this function
- 0314 `virtual_static_not_allowed`:
only nonstatic member functions may be virtual
- 0315 `unqual_function_with_qual_object`:
the object has type qualifiers that are not compatible with the member function
- 0316 `too_many_virtual_functions`:
program too large to compile (too many virtual functions)
- 0317 `bad_return_type_on_virtual_function_override`:
return type is not identical to nor covariant with return type "*type*" of overridden virtual function *entity-kind* "*entity*"
- 0318 `ambiguous_virtual_function_override`:
override of virtual *entity-kind* "*entity*" is ambiguous
- 0319 `pure_specifier_on_nonvirtual_function`:
pure specifier ("= 0") allowed only on virtual functions
- 0320 `bad_pure_specifier`:
badly-formed pure specifier (only "= 0" is allowed)
- 0321 `bad_data_member_initialization`:
data member initializer is not allowed
- 0322 `abstract_class_object_not_allowed`:
object of abstract class type "*type*" is not allowed:
- 0323 `function_returning_abstract_class`:
function returning abstract class "*type*" is not allowed:

- 0324 `duplicate_friend_decl:`
duplicate friend declaration
- 0325 `inline_and_nonfunction:`
inline specifier allowed on function declarations only
- 0326 `inline_not_allowed:`
"inline" is not allowed
- 0327 `bad_storage_class_with_inline:`
invalid storage class for an inline function
- 0328 `bad_member_storage_class:`
invalid storage class for a class member
- 0329 `local_class_function_def_missing:`
local class member *entity-kind* "entity" requires a definition
- 0330 `inaccessible_special_function:`
entity-kind "entity" is inaccessible
- 0332 `missing_const_copy_constructor:`
class "type" has no copy constructor to copy a const object
- 0333 `definition_of_implicitly_declared_function:`
defining an implicitly declared member function is not allowed
- 0334 `no_suitable_copy_constructor:`
class "type" has no suitable copy constructor
- 0335 `linkage_specifier_not_allowed:`
linkage specification is not allowed
- 0336 `bad_linkage_specifier:`
unknown external linkage specification
- 0337 `incompatible_linkage_specifier:`
linkage specification is incompatible with previous "entity"
(declared at line xxx)

- 0338 overloaded_function_linkage:
more than one instance of overloaded function "*entity*" has "C" linkage
- 0339 ambiguous_default_constructor:
class "*type*" has more than one default constructor
- 0340 temp_used_for_ref_init:
value copied to temporary, reference to temporary used
- 0341 nonmember_operator_not_allowed:
"operator:xxxx" must be a member function
- 0342 static_member_operator_not_allowed:
operator may not be a static member function
- 0343 too_many_args_for_conversion:
no arguments allowed on user-defined conversion
- 0344 too_many_args_for_operator:
too many parameters for this operator function
- 0345 too_few_args_for_operator:
too few parameters for this operator function
- 0346 no_params_with_class_type:
nonmember operator requires a parameter with class type
- 0347 default_arg_expr_not_allowed:
default argument is not allowed
- 0348 ambiguous_user_defined_conversion:
more than one user-defined conversion from "*type*" to "*type*" applies:
- 0349 no_matching_operator_function:
no operator "xxxx" matches these operands
- 0350 ambiguous_operator_function:
more than one operator "xxxx" matches these operands:

- 0351 `bad_arg_type_for_operator_new`:
first parameter of allocation function must be of type "size_t"
- 0352 `bad_return_type_for_op_new`:
allocation function requires "void *" return type
- 0353 `bad_return_type_for_op_delete`:
deallocation function requires "void" return type
- 0354 `bad_first_arg_type_for_operator_delete`:
first parameter of deallocation function must be of type "void *"
- 0356 `type_must_be_object_type`:
type must be an object type
- 0357 `base_class_already_initialized`:
base class "*type*" has already been initialized
- 0358 `base_class_init_anachronism`:
base class name required — "*type*" assumed (anachronism)
- 0359 `member_already_initialized`:
entity-kind "*entity*" has already been initialized
- 0360 `missing_base_class_or_member_name`:
name of member or base class is missing
- 0361 `assignment_to_this`:
assignment to "this" (anachronism)
- 0362 `overload_anachronism`:
"overload" keyword used (anachronism)
- 0363 `anon_union_member_access`:
invalid anonymous union — nonpublic member is not allowed
- 0364 `anon_union_member_function`:
invalid anonymous union — member function is not allowed

- 0365 `anon_union_storage_class`:
anonymous union at global or namespace scope must be declared static
- 0366 `missing_initializer_on_fields`:
entity-kind "entity" provides no initializer for:
- 0367 `cannot_initialize_fields`:
implicitly generated constructor for class *"type"* cannot initialize:
- 0368 `no_ctor_but_const_or_ref_member`:
entity-kind "entity" defines no constructor to initialize the following:
- 0369 `var_with_uninitialized_member`:
entity-kind "entity" has an uninitialized const or reference member
- 0370 `var_with_uninitialized_field`:
entity-kind "entity" has an uninitialized const field
- 0371 `missing_const_assignment_operator`:
class *"type"* has no assignment operator to copy a const object
- 0372 `no_suitable_assignment_operator`:
class *"type"* has no suitable assignment operator
- 0373 `ambiguous_assignment_operator`:
ambiguous assignment operator for class *"type"*
- 0375 `missing_typedef_name`:
declaration requires a typedef name
- 0377 `virtual_not_allowed`:
"virtual" is not allowed
- 0378 `static_not_allowed`:
"static" is not allowed
- 0379 `bound_function_cast_anachronism`:
cast of bound function to normal function pointer (anachronism)

- 0380 `expr_not_ptr_to_member`:
expression must have pointer-to-member type
- 0381 `extra_semicolon`:
extra ";" ignored
- 0382 `nonstd_const_member`:
nonstandard member constant declaration (standard form is a static const integral member)
- 0384 `no_matching_new_function`:
no instance of overloaded "*entity*" matches the argument list
- 0386 `no_match_for_addr_of_overloaded_function`:
no instance of *entity-kind* "*entity*" matches the required type
- 0387 `delete_count_anachronism`:
delete array size expression used (anachronism)
- 0388 `bad_return_type_for_op_arrow`:
"operator->" for class "*type*" returns invalid type "*type*"
- 0389 `cast_to_abstract_class`:
a cast to abstract class "*type*" is not allowed:
- 0390 `bad_use_of_main`:
function "main" may not be called or have its address taken
- 0391 `initializer_not_allowed_on_array_new`:
a new-initializer may not be specified for an array
- 0392 `member_function_redecl_outside_class`:
member function "*entity*" may not be redeclared outside its class
- 0393 `ptr_to_incomplete_class_type_not_allowed`:
pointer to incomplete class type is not allowed
- 0394 `ref_to_nested_function_var`:
reference to local variable of enclosing function is not allowed

- 0395 `single_arg_postfix_incr_decr_anachronism`:
single-argument function used for postfix *"xxx"* (anachronism)
- 0397 `bad_default_assignment`:
implicitly generated assignment operator cannot copy:
- 0398 `nonstd_array_cast`:
cast to array type is nonstandard (treated as cast to *"type"*)
- 0399 `class_with_op_new_but_no_op_delete`:
entity-kind "entity" has an operator `newxxxx()` but no default operator `deletexxxx()`
- 0400 `class_with_op_delete_but_no_op_new`:
entity-kind "entity" has a default operator `deletexxxx()` but no operator `newxxxx()`
- 0401 `base_class_with_nonvirtual_dtor`:
destructor for base class *"type"* is not virtual
- 0403 `member_function_redeclaration`:
entity-kind "entity" has already been declared
- 0404 `inline_main`:
function *"main"* may not be declared inline
- 0405 `class_and_member_function_name_conflict`:
member function with the same name as its class must be a constructor
- 0406 `nested_class_anachronism`:
using nested *entity-kind "entity"* (anachronism)
- 0407 `too_many_params_for_destructor`:
a destructor may not have parameters
- 0408 `bad_constructor_param`:
copy constructor for class *"type"* may not have a parameter of type *"type"*

- 0409 incomplete_function_return_type:
entity-kind "entity" returns incomplete type *"type"*
- 0410 protected_access_problem:
protected *entity-kind "entity"* is not accessible through a *"type"* pointer or object
- 0411 param_not_allowed:
a parameter is not allowed
- 0412 asm_decl_not_allowed:
an "asm" declaration is not allowed here
- 0413 no_conversion_function:
no suitable conversion function from *"type"* to *"type"* exists
- 0414 delete_of_incomplete_class:
delete of pointer to incomplete class
- 0415 no_constructor_for_conversion:
no suitable constructor exists to convert from *"type"* to *"type"*
- 0416 ambiguous_constructor_for_conversion:
more than one constructor applies to convert from *"type"* to *"type"*:
- 0417 ambiguous_conversion_function:
more than one conversion function from *"type"* to *"type"* applies:
- 0418 ambiguous_conversion_to_builtin:
more than one conversion function from *"type"* to a built-in type applies:
- 0424 addr_of_constructor_or_destructor:
a constructor or destructor may not have its address taken
- 0425 dollar_used_in_identifier:
dollar sign ("\$\$") used in identifier
- 0426 nonconst_ref_init_anachronism:
temporary used for initial value of reference to non-const (anachronism)

- 0427 `qualifier_in_member_declaration`:
qualified name is not allowed in member declaration
- 0428 `mixed_enum_type_anachronism`:
enumerated type mixed with another type (anachronism)
- 0429 `new_array_size_must_be_nonnegative`:
the size of an array in "new" must be non-negative
- 0430 `return_ref_init_requires_temp`:
returning reference to local temporary
- 0432 `enum_not_allowed`:
"enum" declaration is not allowed
- 0433 `qualifier_dropped_in_ref_init`:
qualifiers dropped in binding reference of type "*type*" to initializer of type "*type*"
- 0434 `bad_nonconst_ref_init`:
a reference of type "*type*" (not const-qualified) cannot be initialized with a value of type "*type*"
- 0435 `delete_of_function_pointer`:
a pointer to function may not be deleted
- 0436 `bad_conversion_function_decl`:
conversion function must be a nonstatic member function
- 0437 `bad_template_declaration_scope`:
template declaration is not allowed here
- 0438 `exp_lt`:
expected a "<"
- 0439 `exp_gt`:
expected a ">"
- 0440 `missing_template_param`:
template parameter declaration is missing

- 0441 `missing_template_arg_list`:
argument list for *entity-kind* "entity" is missing
- 0442 `too_few_template_args`:
too few arguments for *entity-kind* "entity"
- 0443 `too_many_template_args`:
too many arguments for *entity-kind* "entity"
- 0445 `not_used_in_template_function_params`:
entity-kind "entity" is not used in declaring the parameter types of
entity-kind "entity"
- 0446 `cfront_multiple_nested_types`:
two nested types have the same name: "entity" and "entity"
(declared at line .xxxx) (cfront compatibility)
- 0447 `cfront_global_defined_after_nested_type`:
global "entity" was declared after nested "entity" (declared at line
.xxxx) (cfront compatibility)
- 0449 `ambiguous_ptr_to_overloaded_function`:
more than one instance of *entity-kind* "entity" matches the required
type
- 0450 `nonstd_long_long`:
the type "long long" is nonstandard
- 0451 `nonstd_friend_decl`:
omission of "xxxx" is nonstandard
- 0452 `return_type_on_conversion_function`:
return type may not be specified on a conversion function
- 0456 `runaway_recursive_instantiation`:
excessive recursion at instantiation of *entity-kind* "entity"
- 0457 `bad_template_declaration`:
"xxxx" is not a function or static data member

- 0458 `bad_nontype_template_arg`:
argument of type "*type*" is incompatible with template parameter of type "*type*"
- 0459 `init_needing_temp_not_allowed`:
initialization requiring a temporary or conversion is not allowed
- 0460 `decl_hides_function_parameter`:
declaration of "xxxx" hides function parameter
- 0461 `nonconst_ref_init_from_rvalue`:
initial value of reference to non-const must be an lvalue
- 0463 `template_not_allowed`:
"template" is not allowed
- 0464 `not_a_class_template`:
"*type*" is not a class template
- 0466 `function_template_named_main`:
"main" is not a valid name for a function template
- 0467 `union_nonunion_mismatch`:
invalid reference to *entity-kind* "*entity*" (union/nonunion mismatch)
- 0468 `local_type_in_template_arg`:
a template argument may not reference a local type
- 0469 `tag_kind_incompatible_with_declaration`:
tag kind of xxxx is incompatible with declaration of *entity-kind* "*entity*" (declared at line xxxx)
- 0470 `name_not_tag_in_file_scope`:
the global scope has no tag named "xxxx"
- 0471 `not_a_tag_member`:
entity-kind "*entity*" has no tag member named "xxxx"
- 0472 `ptr_to_member_typedef`:
member function typedef (allowed for cfront compatibility)

- 0473 `bad_use_of_member_function_typedef`:
entity-kind "entity" may be used only in pointer-to-member declaration
- 0475 `nonexternal_entity_in_template_arg`:
a template argument may not reference a non-external entity
- 0476 `id_must_be_class_or_type_name`:
name followed by "::~" must be a class name or a type name
- 0477 `destructor_name_mismatch`:
destructor name does not match name of class *"type"*
- 0478 `destructor_type_mismatch`:
type used as destructor name does not match type *"type"*
- 0479 `called_function_redeclared_inline`:
entity-kind "entity" redeclared "inline" after being called
- 0481 `bad_storage_class_on_template_decl`:
invalid storage class for a template declaration
- 0482 `no_access_to_type_cfront_mode`:
entity-kind "entity" is an inaccessible type (allowed for cfront compatibility)
- 0484 `invalid_instantiation_argument`:
invalid explicit instantiation declaration
- 0485 `not_instantiatable_entity`:
entity-kind "entity" is not an entity that can be instantiated
- 0486 `compiler_generated_function_cannot_be_instantiated`:
compiler generated *entity-kind "entity"* cannot be explicitly instantiated
- 0487 `inline_function_cannot_be_instantiated`:
inline *entity-kind "entity"* cannot be explicitly instantiated
- 0488 `pure_virtual_function_cannot_be_instantiated`:
pure virtual *entity-kind "entity"* cannot be explicitly instantiated

- 0489 instantiation_requested_no_definition_supplied:
entity-kind "entity" cannot be instantiated — no template definition was supplied
- 0490 instantiation_requested_and_specialized:
entity-kind "entity" cannot be instantiated — it has been explicitly specialized
- 0491 no_constructor:
 class *"type"* has no constructor
- 0493 no_match_for_type_of_overloaded_function:
 no instance of *entity-kind "entity"* matches the specified type
- 0494 nonstd_void_param_list:
 declaring a void parameter list with a typedef is nonstandard
- 0495 cfront_name_lookup_bug:
 global *entity-kind "entity"* used instead of *entity-kind "entity"* (cfront compatibility)
- 0496 redeclaration_of_template_param_name:
 template parameter *"xxxx"* may not be redeclared in this scope
- 0497 decl_hides_template_parameter:
 declaration of *"xxxx"* hides template parameter
- 0498 must_be_prototype_instantiation:
 template argument list must match the parameter list
- 0500 bad_extra_arg_for_postfix_operator:
 extra parameter of postfix *"operatorxxxx"* must be of type *"int"*
- 0501 function_type_required:
 an operator name must be declared as a function
- 0502 operator_name_not_allowed:
 operator name is not allowed
- 0503 bad_scope_for_specialization:
entity-kind "entity" cannot be specialized in the current scope

- 0504 `nonstd_member_function_address`:
nonstandard form for taking the address of a member function
- 0505 `too_few_template_params`:
too few template parameters — does not match previous declaration
- 0506 `too_many_template_params`:
too many template parameters — does not match previous declaration
- 0507 `template_operator_delete`:
function template for operator delete(void *) is not allowed
- 0508 `class_template_same_name_as_tmpl_param`:
class template and template parameter may not have the same name
- 0510 `unnamed_type_in_template_arg`:
a template argument may not reference an unnamed type
- 0511 `enum_type_not_allowed`:
enumerated type is not allowed
- 0512 `qualified_reference_type`:
type qualifier on a reference type is not allowed
- 0513 `incompatible_assignment_operands`:
a value of type "*type*" cannot be assigned to an entity of type "*type*"
- 0514 `unsigned_compare_with_negative`:
pointless comparison of unsigned integer with a negative constant
- 0515 `converting_to_incomplete_class`:
cannot convert to incomplete class "*type*"
- 0516 `missing_initializer_on_unnamed_const`:
const object requires an initializer
- 0517 `unnamed_object_with_uninitialized_field`:
object has an uninitialized const or reference member

- 0518 `nonstd_pp_directive`:
nonstandard preprocessing directive
- 0519 `unexpected_template_arg_list`:
entity-kind "entity" may not have a template argument list
- 0520 `missing_initializer_list`:
initialization with "{...}" expected for aggregate object
- 0521 `incompatible_ptr_to_member_selection_operands`:
pointer-to-member selection class types are incompatible ("*type*" and "*type*")
- 0522 `self_friendship`:
pointless friend declaration
- 0523 `period_used_as_qualifier`:
"." used in place of "::" to form a qualified name (cfront anachronism)
- 0524 `const_function_anachronism`:
non-const function called for const object (anachronism)
- 0525 `dependent_stmt_is_declaration`:
a dependent statement may not be a declaration
- 0526 `void_param_not_allowed`:
a parameter may not have void type
- 0529 `bad_tmpl_arg_expr_operator`:
this operator is not allowed in a template argument expression
- 0530 `missing_handler`:
try block requires at least one handler
- 0531 `missing_exception_declaration`:
handler requires an exception declaration
- 0532 `masked_by_default_handler`:
handler is masked by default handler

- 0533 `masked_by_handler`:
handler is potentially masked by previous handler for type *"type"*
- 0534 `local_type_used_in_exception`:
use of a local type to specify an exception
- 0535 `redundant_exception_specification_type`:
redundant type in exception specification
- 0536 `incompatible_exception_specification`:
exception specification is incompatible with that of previous
entity-kind "entity" (declared at line *xxxx*):
- 0540 `no_exception_support`:
support for exception handling is disabled
- 0541 `omitted_exception_specification`:
omission of exception specification is incompatible with previous
entity-kind "entity" (declared at line *xxxx*)
- 0542 `cannot_create_instantiation_request_file`:
could not create instantiation request file *"xxxx"*
- 0543 `non_arith_operation_in_tmpl_arg`:
non-arithmetic operation not allowed in nontype template
argument
- 0544 `local_type_in_nonlocal_var`:
use of a local type to declare a nonlocal variable
- 0545 `local_type_in_function`:
use of a local type to declare a function
- 0546 `branch_past_initialization`:
transfer of control bypasses initialization of:
- 0548 `branch_into_handler`:
transfer of control into an exception handler
- 0549 `used_before_set`:
entity-kind "entity" is used before its value is set

- 0550 `set_but_not_used`:
entity-kind "entity" was set but never used
- 0551 `bad_scope_for_definition`:
entity-kind "entity" cannot be defined in the current scope
- 0552 `exception_specification_not_allowed`:
exception specification is not allowed
- 0553 `template_and_instance_linkage_conflict`:
external/internal linkage conflict for *entity-kind "entity"* (declared at line xxxx)
- 0554 `conversion_function_not_usable`:
entity-kind "entity" will not be called for implicit or explicit conversions
- 0555 `tag_kind_incompatible_with_template_parameter`:
tag kind of xxxx is incompatible with template parameter of type *"type"*
- 0556 `template_operator_new`:
function template for operator `new(size_t)` is not allowed
- 0558 `bad_member_type_in_ptr_to_member`:
pointer to member of type *"type"* is not allowed
- 0559 `ellipsis_on_operator_function`:
ellipsis is not allowed in operator function parameter list
- 0560 `unimplemented_keyword`:
"entity" is reserved for future use as a keyword
- 0561 `cl_invalid_macro_definition`:
invalid macro definition:
- 0562 `cl_invalid_macro_undefinition`:
invalid macro undefinition:
- 0563 `cl_invalid_preprocessor_output_file`:
invalid preprocessor output file

- 0564 cl_cannot_open_preprocessor_output_file:
cannot open preprocessor output file
- 0565 cl_il_file_must_be_specified:
IL file name must be specified if input is
- 0566 cl_invalid_il_output_file:
invalid IL output file
- 0567 cl_cannot_open_il_output_file:
cannot open IL output file
- 0568 cl_invalid_C_output_file:
invalid C output file
- 0569 cl_cannot_open_C_output_file:
cannot open C output file
- 0570 cl_error_in_debug_option_argument:
error in debug option argument
- 0571 cl_invalid_option:
invalid option:
- 0572 cl_back_end_requires_il_file:
back end requires name of IL file
- 0573 cl_could_not_open_il_file:
could not open IL file
- 0574 cl_invalid_number:
invalid number:
- 0575 cl_incorrect_host_id:
incorrect host CPU id
- 0576 cl_invalid_instantiation_mode:
invalid instantiation mode:
- 0578 cl_invalid_error_limit:
invalid error limit:

- 0579 `cl_invalid_raw_listing_output_file:`
invalid raw-listing output file
- 0580 `cl_cannot_open_raw_listing_output_file:`
cannot open raw-listing output file
- 0581 `cl_invalid_xref_output_file:`
invalid cross-reference output file
- 0582 `cl_cannot_open_xref_output_file:`
cannot open cross-reference output file
- 0583 `cl_invalid_error_output_file:`
invalid error output file
- 0584 `cl_cannot_open_error_output_file:`
cannot open error output file
- 0585 `cl_vtbl_option_only_in_cplusplus:`
virtual function tables can only be suppressed when compiling C++
- 0586 `cl_anachronism_option_only_in_cplusplus:`
anachronism option can be used only when compiling C++
- 0587 `cl_instantiation_option_only_in_cplusplus:`
instantiation mode option can be used only when compiling C++
- 0588 `cl_auto_instantiation_option_only_in_cplusplus:`
automatic instantiation mode can be used only when compiling C++
- 0589 `cl_implicit_inclusion_option_only_in_cplusplus:`
implicit template inclusion mode can be used only when compiling C++
- 0590 `cl_exceptions_option_only_in_cplusplus:`
exception handling option can be used only when compiling C++
- 0591 `cl_strict_ansi_incompatible_with_pcc:`
strict ANSI mode is incompatible with K&R mode

- 0592 `cl_strict_ansi_incompatible_with_cfront`:
strict ANSI mode is incompatible with cfront mode
- 0593 `cl_missing_source_file_name`:
missing source file name
- 0594 `cl_output_file_incompatible_with_multiple_inputs`:
output files may not be specified when compiling several input files
- 0595 `cl_too_many_arguments`:
too many arguments on command line
- 0596 `cl_no_output_file_needed`:
an output file was specified, but none is needed
- 0597 `cl_il_display_requires_il_file_name`:
IL display requires name of IL file
- 0598 `void_template_parameter`:
a template parameter may not have void type
- 0599 `too_many_unused_instantiations`:
excessive recursive instantiation of *entity-kind "entity"* due to
instantiate-all mode
- 0600 `cl_strict_ansi_incompatible_with_anachronisms`:
strict ANSI mode is incompatible with allowing anachronisms
- 0601 `void_throw`:
a throw expression may not have void type
- 0602 `cl_tim_local_conflicts_with_auto_instantiation`:
local instantiation mode is incompatible with automatic instantiation
- 0603 `abstract_class_param_type`:
parameter of abstract class type *"type"* is not allowed:
- 0604 `array_of_abstract_class`:
array of abstract class *"type"* is not allowed:

- 0605 `float_template_parameter`:
floating-point template parameter is nonstandard
- 0606 `pragma_must_precede_declaration`:
this pragma must immediately precede a declaration
- 0607 `pragma_must_precede_statement`:
this pragma must immediately precede a statement
- 0608 `pragma_must_precede_decl_or_stmt`:
this pragma must immediately precede a declaration or statement
- 0609 `pragma_may_not_be_used_here`:
this kind of pragma may not be used here
- 0611 `partial_override`:
overloaded virtual function "*entity*" is only partially overridden in *entity-kind* "*entity*"
- 0612 `specialization_of_called_inline_template_function`:
specific definition of inline template function must precede its first use
- 0613 `cl_invalid_error_tag`:
invalid error tag:
- 0614 `cl_invalid_error_number`:
invalid error number:
- 0615 `param_type_ptr_to_array_of_unknown_bound`:
parameter type involves pointer to array of unknown bound
- 0616 `param_type_ref_array_of_unknown_bound`:
parameter type involves reference to array of unknown bound
- 0617 `ptr_to_member_cast_to_ptr_to_function`:
pointer-to-member-function cast to pointer to function
- 0618 `no_named_fields`:
struct or union declares no named members

- 0619 `nonstd_unnamed_field:`
nonstandard unnamed field
- 0620 `nonstd_unnamed_member:`
nonstandard unnamed member
- 0622 `cl_invalid_pch_output_file:`
invalid precompiled header output file
- 0623 `cl_cannot_open_pch_output_file:`
cannot open precompiled header output file
- 0624 `not_a_type_name:`
"xxxx" is not a type name
- 0625 `cl_cannot_open_pch_input_file:`
cannot open precompiled header input file
- 0626 `invalid_pch_file:`
precompiled header file "xxxx" is either invalid or not generated by
this version of the compiler
- 0627 `pch_curr_directory_changed:`
precompiled header file "xxxx" was not generated in this directory
- 0628 `pch_header_files_have_changed:`
header files used to generate precompiled header file "xxxx" have
changed
- 0629 `pch_cmd_line_option_mismatch:`
the command line options do not match those used when
precompiled header file "xxxx" was created
- 0630 `pch_file_prefix_mismatch:`
the initial sequence of preprocessing directives is not compatible
with those of precompiled header file "xxxx"
- 0631 `unable_to_get_mapped_memory:`
unable to obtain mapped memory

- 0632 using_pch:
"xxxx": using precompiled header file "xxxx"
- 0633 creating_pch:
"xxxx": creating precompiled header file "xxxx"
- 0634 memory_mismatch:
memory usage conflict with precompiled header file "xxxx"
- 0635 cl_invalid_pch_size:
invalid PCH memory size
- 0636 cl_pch_must_be_first:
PCH options must appear first in the command line
- 0637 out_of_memory_during_pch_allocation:
insufficient memory for PCH memory allocation
- 0638 cl_pch_incompatible_with_multiple_inputs:
precompiled header files may not be used when compiling several input files
- 0639 not_enough_preallocated_memory:
insufficient preallocated memory for generation of precompiled header file (xxxx bytes required)
- 0640 program_entity_too_large_for_pch:
very large entity in program prevents generation of precompiled header file
- 0641 cannot_chdir:
"xxxx" is not a valid directory
- 0642 cannot_build_temp_file_name:
cannot build temporary file name
- 0643 restrict_not_allowed:
"restrict" is not allowed

- 0644 `restrict_pointer_to_function`:
a pointer or reference to function type may not be qualified by
"restrict"
- 0645 `bad_declspec_modifier`:
"xxxx" is an unrecognized `__declspec` attribute
- 0646 `calling_convention_not_allowed`:
a calling convention modifier may not be specified here
- 0647 `conflicting_calling_conventions`:
conflicting calling convention modifiers
- 0648 `cl_strict_ansi_incompatible_with_microsoft`:
strict ANSI mode is incompatible with Microsoft mode
- 0649 `cl_cfront_incompatible_with_microsoft`:
cfront mode is incompatible with Microsoft mode
- 0650 `calling_convention_ignored`:
calling convention specified here is ignored
- 0651 `calling_convention_may_not_precede_nested_declarator`:
a calling convention may not be followed by a nested declarator
- 0652 `calling_convention_ignored_for_type`:
calling convention is ignored for this type
- 0654 `decl_modifiers_incompatible_with_previous_decl`:
declaration modifiers are incompatible with previous declaration
- 0655 `decl_modifiers_invalid_for_this_decl`:
the modifier "xxxx" is not allowed on this declaration
- 0656 `branch_into_try_block`:
transfer of control into a try block
- 0657 `incompatible_inline_specifier_on_specific_decl`:
inline specification is incompatible with previous "entity" (declared
at line xxxx)

- 0658 `template_missing_closing_brace`:
closing brace of template definition not found
- 0659 `cl_wchar_t_option_only_in_cplusplus`:
`wchar_t` keyword option can be used only when compiling C++
- 0660 `bad_pack_alignment`:
invalid packing alignment value
- 0661 `exp_int_constant`:
expected an integer constant
- 0662 `call_of_pure_virtual`:
call of pure virtual function
- 0663 `bad_ident_string`:
invalid source file identifier string
- 0664 `template_friend_definition_not_allowed`:
a class template cannot be defined in a friend declaration
- 0665 `asm_not_allowed`:
"asm" is not allowed
- 0666 `bad_asm_function_def`:
"asm" must be used with a function definition
- 0667 `nonstd_asm_function`:
"asm" function is nonstandard
- 0668 `nonstd_ellipsis_only_param`:
ellipsis with no explicit parameters is nonstandard
- 0669 `nonstd_address_of_ellipsis`:
"&..." is nonstandard
- 0670 `bad_address_of_ellipsis`:
invalid use of "&..."

- 0672 `const_volatile_ref_init_anachronism`:
temporary used for initial value of reference to const volatile
(anachronism)
- 0673 `bad_const_volatile_ref_init`:
a reference of type "*type*" cannot be initialized with a value of type
"*type*"
- 0674 `const_volatile_ref_init_from_rvalue`:
initial value of reference to const volatile must be an lvalue
- 0675 `cl_SVR4_C_option_only_in_ansi_C`:
SVR4 C compatibility option can be used only when compiling ANSI
C
- 0676 `using_out_of_scope_declaration`:
using out-of-scope declaration of *entity-kind* "*entity*" (declared at
line *xxxx*)
- 0677 `cl_strict_ansi_incompatible_with_SVR4`:
strict ANSI mode is incompatible with SVR4 C mode
- 0678 `cannot_inline_call`:
call of *entity-kind* "*entity*" (declared at line *xxxx*) cannot be inlined
- 0679 `cannot_inline`:
entity-kind "*entity*" cannot be inlined
- 0680 `cl_invalid_pch_directory`:
invalid PCH directory:
- 0681 `exp_except_or_finally`:
expected `__except` or `__finally`
- 0682 `leave_must_be_in_try`:
a `__leave` statement may only be used within a `__try`
- 0688 `not_found_on_pack_alignment_stack`:
"*xxxx*" not found on pack alignment stack

- 0689 `empty_pack_alignment_stack`:
empty pack alignment stack
- 0690 `cl_rtti_option_only_in_cplusplus`:
RTTI option can be used only when compiling C++
- 0691 `inaccessible_elided_ctor`:
entity-kind "entity", required for copy that was eliminated, is inaccessible
- 0692 `uncallable_elided_ctor`:
entity-kind "entity", required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue
- 0693 `typeid_needs_typeinfo`:
<typeid> must be included before typeid is used
- 0694 `cannot_cast_away_const`:
xxxx cannot cast away const or other type qualifiers
- 0695 `bad_dynamic_cast_type`:
the type in a `dynamic_cast` must be a pointer or reference to a complete class type, or void *
- 0696 `bad_ptr_dynamic_cast_operand`:
the operand of a pointer `dynamic_cast` must be a pointer to a complete class type
- 0697 `bad_ref_dynamic_cast_operand`:
the operand of a reference `dynamic_cast` must be an lvalue of a complete class type
- 0698 `dynamic_cast_operand_must_be_polymorphic`:
the operand of a runtime `dynamic_cast` must have a polymorphic class type
- 0699 `cl_bool_option_only_in_cplusplus`:
bool option can be used only when compiling C++
- 0701 `array_type_not_allowed`:
an array type is not allowed here

- 0702 exp_assign:
expected an "="
- 0703 exp_declarator_in_condition_decl:
expected a declarator in condition declaration
- 0704 redeclaration_of_condition_decl_name:
"xxxx", declared in condition, may not be redeclared in this scope
- 0705 default_template_arg_not_allowed:
default template arguments are not allowed for function templates
- 0706 exp_comma_or_gt:
expected a ",", or ">"
- 0707 missing_template_param_list:
expected a template parameter list
- 0708 incr_of_bool_deprecated:
incrementing a bool value is deprecated
- 0709 bool_type_not_allowed:
bool type is not allowed
- 0710 base_class_offset_too_large:
offset of base class "*entity*" within class "*entity*" is too large
- 0711 expr_not_bool:
expression must have bool type (or be convertible to bool)
- 0712 cl_array_new_and_delete_option_only_in_cplusplus:
array new and delete option can be used only when compiling C++
- 0713 based_requires_variable_name:
entity-kind "*entity*" is not a variable name
- 0714 based_not_allowed_here:
__based modifier is not allowed here
- 0715 based_not_followed_by_star:
__based does not precede a pointer operator, __based ignored

- 0716 `based_var_must_be_ptr`:
variable in `__based` modifier must have pointer type
- 0717 `bad_const_cast_type`:
the type in a `const_cast` must be a pointer, reference, or pointer to member to an object type
- 0718 `bad_const_cast`:
a `const_cast` can only adjust type qualifiers; it cannot change the underlying type
- 0719 `mutable_not_allowed`:
mutable is not allowed
- 0720 `cannot_change_access`:
redeclaration of *entity-kind* "*entity*" is not allowed to alter its access
- 0721 `nonstd_printf_format_string`:
nonstandard format string conversion
- 0722 `probable_inadvertent_lbracket_digraph`:
use of alternative token "<:" appears to be unintended
- 0723 `probable_inadvertent_sharp_digraph`:
use of alternative token "%:" appears to be unintended
- 0724 `namespace_def_not_allowed`:
namespace definition is not allowed
- 0725 `missing_namespace_name`:
name must be a namespace name
- 0726 `namespace_alias_def_not_allowed`:
namespace alias definition is not allowed
- 0727 `namespace_qualified_name_required`:
namespace-qualified name is required
- 0728 `namespace_name_not_allowed`:
a namespace name is not allowed

- 0729 `bad_combination_of_dll_attributes`:
invalid combination of DLL attributes
- 0730 `sym_not_a_class_template`:
entity-kind "entity" is not a class template
- 0731 `array_of_incomplete_type`:
array with incomplete element type is nonstandard
- 0732 `allocation_operator_in_namespace`:
allocation operator may not be declared in a namespace
- 0733 `deallocation_operator_in_namespace`:
deallocation operator may not be declared in a namespace
- 0734 `conflicts_with_using_decl`:
entity-kind "entity" conflicts with using-declaration of *entity-kind "entity"*
- 0735 `using_decl_conflicts_with_prev_decl`:
using-declaration of *entity-kind "entity"* conflicts with *entity-kind "entity"* (declared at line `xxxx`)
- 0736 `cl_namespaces_option_only_in_cplusplus`:
namespaces option can be used only when compiling C++
- 0737 `useless_using_declaration`:
using-declaration ignored — it refers to the current namespace
- 0738 `class_qualified_name_required`:
a class-qualified name is required
- 0741 `using_declaration_ignored`:
using-declaration of *entity-kind "entity"* ignored
- 0742 `not_an_actual_member`:
entity-kind "entity" has no actual member `xxxx`
- 0744 `mem_attrib_incompatible`:
incompatible memory attributes specified

- 0745 `mem_attr_ignored`:
memory attribute ignored
- 0746 `mem_attr_may_not_precede_nested_declarator`:
memory attribute may not be followed by a nested declarator
- 0747 `dupl_mem_attr`:
memory attribute specified more than once
- 0748 `dupl_calling_convention`:
calling convention specified more than once
- 0749 `type_qualifier_not_allowed`:
a type qualifier is not allowed
- 0750 `template_instance_already_used`:
entity-kind "entity" (declared at line xxxx) was used before its
template was declared
- 0751 `static_nonstatic_with_same_param_types`:
static and nonstatic member functions with same parameter types
cannot be overloaded
- 0752 `no_prior_declaration`:
no prior declaration of *entity-kind "entity"*
- 0753 `template_id_not_allowed`:
a template-id is not allowed
- 0754 `class_qualified_name_not_allowed`:
a class-qualified name is not allowed
- 0755 `bad_scope_for_redeclaration`:
entity-kind "entity" may not be redeclared in the current scope
- 0756 `qualifier_in_namespace_member_decl`:
qualified name is not allowed in namespace member declaration
- 0757 `sym_not_a_type_name`:
entity-kind "entity" is not a type name

- 0758 `explicit_instantiation_not_in_namespace_scope`:
explicit instantiation is not allowed in the current scope
- 0759 `bad_scope_for_explicit_instantiation`:
entity-kind "entity" cannot be explicitly instantiated in the current scope
- 0760 `multiple_explicit_instantiations`:
entity-kind "entity" explicitly instantiated more than once
- 0761 `typename_not_in_template`:
typename may only be used within a template
- 0762 `cl_special_subscript_cost_option_only_in_cplusplus`:
special_subscript_cost option can be used only when compiling C++
- 0763 `cl_typename_option_only_in_cplusplus`:
typename option can be used only when compiling C++
- 0764 `cl_implicit_typename_option_only_in_cplusplus`:
implicit typename option can be used only when compiling C++
- 0765 `nonstd_character_at_start_of_macro_def`:
nonstandard character at start of object-like macro definition
- 0766 `exception_spec_override_incompat`:
exception specification for virtual *entity-kind "entity"* is incompatible with that of overridden *entity-kind "entity"*
- 0767 `pointer_conversion_loses_bits`:
conversion from pointer to smaller integer
- 0768 `generated_exception_spec_override_incompat`:
exception specification for implicitly declared virtual *entity-kind "entity"* is incompatible with that of overridden *entity-kind "entity"*
- 0769 `implicit_call_of_ambiguous_name`:
"entity", implicitly called from *entity-kind "entity"*, is ambiguous

- 0770 `cl_explicit_option_only_in_cplusplus`:
option "explicit" can be used only when compiling C++
- 0771 `explicit_not_allowed`:
"explicit" is not allowed
- 0772 `conflicts_with_predeclared_type_info`:
declaration conflicts with "xxxx" (reserved class name)
- 0773 `array_member_initialization`:
only "()" is allowed as initializer for array *entity-kind* "entity"
- 0774 `virtual_function_template`:
"virtual" is not allowed in a function template declaration
- 0775 `anon_union_class_member_template`:
invalid anonymous union — class member template is not allowed
- 0776 `template_depth_mismatch`:
template nesting depth does not match the previous declaration of *entity-kind* "entity"
- 0777 `multiple_template_decls_not_allowed`:
this declaration cannot have multiple "template <...>" clauses
- 0778 `cl_old_for_init_option_only_in_cplusplus`:
option to control the for-init scope can be used only when compiling C++
- 0779 `redeclaration_of_for_init_decl_name`:
"xxxx", declared in for-loop initialization, may not be redeclared in this scope
- 0780 `hidden_by_old_for_init`:
reference is to *entity-kind* "entity" (declared at line xxxx) — under old for-init scoping rules it would have been *entity-kind* "entity" (declared at line xxxx)
- 0781 `cl_for_init_diff_warning_option_only_in_cplusplus`:
option to control warnings on for-init differences can be used only when compiling C++

- 0782 `unnamed_class_virtual_function_def_missing`:
definition of virtual *entity-kind* "entity" is required here
- 0783 `svr4_token_pasting_comment`:
empty comment interpreted as token-pasting operator "##"
- 0784 `storage_class_in_friend_decl`:
a storage class is not allowed in a friend declaration
- 0785 `templ_param_list_not_allowed`:
template parameter list for "entity" is not allowed in this declaration
- 0786 `bad_member_template_sym`:
entity-kind "entity" is not a valid member class or function template
- 0787 `bad_member_template_decl`:
not a valid member class or function template declaration
- 0788 `specialization_follows_param_list`:
a template declaration containing a template parameter list may not be followed by an explicit specialization declaration
- 0789 `specialization_of_referenced_template`:
explicit specialization of *entity-kind* "entity" must precede the first use of *entity-kind* "entity"
- 0790 `explicit_specialization_not_in_namespace_scope`:
explicit specialization is not allowed in the current scope
- 0791 `partial_specialization_not_allowed`:
partial specialization of *entity-kind* "entity" is not allowed
- 0792 `entity_cannot_be_specialized`:
entity-kind "entity" is not an entity that can be explicitly specialized
- 0793 `specialization_of_referenced_entity`:
explicit specialization of *entity-kind* "entity" must precede its first use

- 0794 `template_param_in_elab_type`:
template parameter `xxxx` may not be used in an elaborated type specifier
- 0795 `old_specialization_not_allowed`:
specializing *entity-kind* "*entity*" requires "template<>" syntax
- 0798 `cl_old_specializations_option_only_in_cplusplus`:
option "old_specializations" can be used only when compiling C++
- 0799 `nonstd_old_specialization`:
specializing *entity-kind* "*entity*" without "template<>" syntax is nonstandard
- 0800 `bad_linkage_for_decl`:
this declaration may not have extern "C" linkage
- 0801 `not_a_template_name`:
"`xxxx`" is not a class or function template name in the current scope
- 0802 `nonstd_default_arg_on_function_template_redecl`:
specifying a default argument when redeclaring an unreferenced function template is nonstandard
- 0803 `default_arg_on_function_template_not_allowed`:
specifying a default argument when redeclaring an already referenced function template is not allowed
- 0804 `pm_derived_class_from_virtual_base`:
cannot convert pointer to member of base class "*type*" to pointer to member of derived class "*type*" — base class is virtual
- 0805 `bad_exception_specification_for_specialization`:
exception specification is incompatible with that of *entity-kind* "*entity*" (declared at line `xxxx`):
- 0806 `omitted_exception_specification_on_specialization`:
omission of exception specification is incompatible with *entity-kind* "*entity*" (declared at line `xxxx`)

- 0807 `unexpected_end_of_default_arg`:
unexpected end of default argument expression
- 0808 `default_init_of_reference`:
default-initialization of reference is not allowed
- 0809 `uninitialized_field_with_const_member`:
uninitialized *entity-kind* "*entity*" has a const member
- 0810 `uninitialized_base_class_with_const_member`:
uninitialized base class "*type*" has a const member
- 0811 `missing_default_constructor_on_const`:
const *entity-kind* "*entity*" requires an initializer — class "*type*" has no explicitly declared default constructor
- 0812 `missing_default_constructor_on_unnamed_const`:
const object requires an initializer — class "*type*" has no explicitly declared default constructor
- 0813 `cl_impl_extern_c_conv_option_only_in_cplusplus`:
option "`implicit_extern_c_type_conversion`" can be used only when compiling C++
- 0814 `cl_strict_ansi_incompatible_with_long_preserving`:
strict ANSI mode is incompatible with long preserving rules
- 0815 `useless_type_qualifier_on_return_type`:
type qualifier on return type is meaningless
- 0816 `type_qualifier_on_void_return_type`:
in a function definition a type qualifier on a "void" return type is not allowed
- 0817 `static_data_member_not_allowed`:
static data member declaration is not allowed in this class
- 0818 `invalid_declaration`:
template instantiation resulted in an invalid function declaration

- 0819 `ellipsis_not_allowed`:
 "..." is not allowed
- 0820 `cl_extern_inline_option_only_in_cplusplus`:
 option "extern_inline" can be used only when compiling C++
- 0821 `extern_inline_never_defined`:
 extern inline *entity-kind "entity"* was referenced but not defined
- 0822 `invalid_destructor_name`:
 invalid destructor name for type *"type"*
- 0824 `ambiguous_destructor`:
 destructor reference is ambiguous — both *entity-kind "entity"* and *entity-kind "entity"* could be used
- 0825 `virtual_inline_never_defined`:
 virtual inline *entity-kind "entity"* was never defined
- 0826 `unreferenced_function_param`:
entity-kind "entity" was never referenced
- 0827 `union_already_initialized`:
 only one member of a union may be specified in a constructor initializer list
- 0828 `no_array_new_and_delete_support`:
 support for "new[]" and "delete[]" is disabled
- 0829 `double_for_long_double`:
 "double" used for "long double" in generated C code
- 0830 `no_corresponding_delete`:
entity-kind "entity" has no corresponding operator `delete` (to be called if an exception is thrown during initialization of an allocated object)
- 0831 `useless_placement_delete`:
 support for placement delete is disabled

- 0832 `no_appropriate_delete`:
no appropriate operator delete is visible
- 0833 `ptr_or_ref_to_incomplete_type`:
pointer or reference to incomplete type is not allowed
- 0834 `bad_partial_specialization`:
invalid partial specialization — *entity-kind* "entity" is already fully specialized
- 0835 `incompatible_exception_specs`:
incompatible exception specifications
- 0836 `returning_ref_to_local_variable`:
returning reference to local variable
- 0837 `nonstd_implicit_int`:
omission of explicit type is nonstandard ("int" assumed)
- 0838 `ambiguous_partial_spec`:
more than one partial specialization matches the template argument list of *entity-kind* "entity"
- 0840 `partial_spec_is_primary_template`:
a template argument list is not allowed in a declaration of a primary template
- 0841 `default_not_allowed_on_partial_spec`:
partial specializations may not have default template arguments
- 0842 `not_used_in_partial_spec_arg_list`:
entity-kind "entity" is not used in template argument list of *entity-kind* "entity"
- 0843 `partial_spec_param_depends_on_tmpl_param`:
the type of partial specialization template parameter *entity-kind* "entity" depends on another template parameter
- 0844 `partial_spec_arg_depends_on_tmpl_param`:
the template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter

- 0845 `partial_spec_after_instantiation`:
this partial specialization would have been used to instantiate *entity-kind "entity"*
- 0846 `partial_spec_after_instantiation_ambiguous`:
this partial specialization would have been made the instantiation of *entity-kind "entity"* ambiguous
- 0847 `expr_not_integral_or_enum`:
expression must have integral or enum type
- 0848 `expr_not_arithmetic_or_enum`:
expression must have arithmetic or enum type
- 0849 `expr_not_arithmetic_or_enum_or_pointer`:
expression must have arithmetic, enum, or pointer type
- 0850 `cast_not_integral_or_enum`:
type of cast must be integral or enum
- 0851 `cast_not_arithmetic_or_enum_or_pointer`:
type of cast must be arithmetic, enum, or pointer
- 0852 `expr_not_object_pointer`:
expression must be a pointer to a complete object type
- 0853 `member_partial_spec_not_in_class`:
a partial specialization of a member class template must be declared in the class of which it is a member
- 0854 `partial_spec_nontype_expr`:
a partial specialization nontype argument must be the name of a nontype parameter or a constant
- 0855 `different_return_type_on_virtual_function_override`:
return type is not identical to return type *"type"* of overridden virtual function *entity-kind "entity"*
- 0856 `cl_guiding_decls_option_only_in_cplusplus`:
option *"guiding_decls"* can be used only when compiling C++

- 0857 `member_partial_spec_not_in_namespace`:
a partial specialization of a class template must be declared in the namespace of which it is a member
- 0858 `pure_virtual_function`:
entity-kind "entity" is a pure virtual function
- 0859 `no_overrider_for_pure_virtual_function`:
pure virtual *entity-kind "entity"* has no overrider
- 0860 `decl_modifiers_ignored`:
`__declspec` attributes ignored
- 0861 `invalid_char`:
invalid character in input line
- 0862 `incomplete_return_type`:
function returns incomplete type "*type*"
- 0863 `local_pragma_pack`:
effect of this "#pragma pack" directive is local to *entity-kind "entity"*
- 0864 `not_a_template`:
`xxxx` is not a template
- 0865 `friend_partial_specialization`:
a friend declaration may not declare a partial specialization
- 0866 `exception_specification_ignored`:
exception specification ignored
- 0867 `unexpected_type_for_size_t`:
declaration of "`size_t`" does not match the expected type "*type*"
- 0868 `exp_gt_not_shift_right`:
space required between adjacent ">" delimiters of nested template argument lists ("`>>`" is the right shift operator)
- 0869 `bad_multibyte_char_locale`:
could not set locale "`xxxx`" to allow processing of multibyte characters

- 0870 `bad_multibyte_char`:
invalid multibyte character sequence
- 0871 `bad_type_from_instantiation`:
template instantiation resulted in unexpected function type of "*type*"
(the meaning of a name may have changed since the template
declaration — the type of the template is "*type*")
- 0872 `ambiguous_guiding_decl`:
ambiguous guiding declaration — more than one function template
"*entity*" matches type "*type*"
- 0873 `non_integral_operation_in_tmpl_arg`:
non-integral operation not allowed in nontype template argument
- 0874 `cl_embedded_cplusplus_option_only_in_cplusplus`:
option "embedded_c++" can be used only when compiling C++
- 0875 `templates_in_embedded_cplusplus`:
Embedded C++ does not support templates
- 0876 `exceptions_in_embedded_cplusplus`:
Embedded C++ does not support exception handling
- 0877 `namespaces_in_embedded_cplusplus`:
Embedded C++ does not support namespaces
- 0878 `rtti_in_embedded_cplusplus`:
Embedded C++ does not support run time type information
- 0879 `new_cast_in_embedded_cplusplus`:
Embedded C++ does not support the new cast syntax
- 0880 `using_decl_in_embedded_cplusplus`:
Embedded C++ does not support using declarations
- 0881 `mutable_in_embedded_cplusplus`:
Embedded C++ does not support "mutable"
- 0882 `multiple_inheritance_in_embedded_cplusplus`:
Embedded C++ does not support multiple or virtual inheritance

- 0883 `cl_invalid_microsoft_version`:
invalid Microsoft version number
- 0884 `inheritance_kind_already_set`:
pointer-to-member representation has already been set for
entity-kind "entity"
- 0885 `bad_constructor_type`:
"type" cannot be used to designate constructor for *"type"*
- 0886 `bad_suffix`:
invalid suffix on integral constant
- 0887 `uuidof_requires_uuid_class_type`:
operand of `__uuidof` must have a class type for which
`__declspec(uuid("..."))` has been specified
- 0888 `bad_uuid_string`:
invalid GUID string in `__declspec(uuid("..."))`
- 0889 `cl_vla_option_only_in_C`:
option "vla" can be used only when compiling C
- 0890 `vla_with_unspecified_bound_not_allowed`:
variable length array with unspecified bound is not allowed
- 0891 `explicit_template_args_not_allowed`:
an explicit template argument list is not allowed on this declaration
- 0892 `variably_modified_type_not_allowed`:
an entity with linkage cannot have a variably modified type
- 0893 `vla_is_not_auto`:
a variable length array cannot have static storage duration
- 0894 `sym_not_a_template`:
entity-kind "entity" is not a template
- 0896 `expected_template_arg`:
expected a template argument

- 0897 `explicit_template_args_in_expr`:
explicit function template argument lists are not supported yet in expression contexts
- 0898 `no_params_with_class_or_enum_type`:
nonmember operator requires a parameter with class or enum type
- 0899 `cl_enum_overloading_option_only_in_cplusplus`:
option "enum_overloading" can be used only when compiling C++
- 0901 `destructor_qualifier_type_mismatch`:
qualifier of destructor name "*type*" does not match type "*type*"
- 0902 `type_qualifier_ignored`:
type qualifier ignored
- 0903 `cl_nonstandard_qualifier_deduction_option_only_in_cplusplus`:
option "nonstd_qualifier_deduction" can be used only when compiling C++
- 0905 `bad_declspec_property`:
incorrect property specification; correct form is
`__declspec(property(get=name1,put=name2))`
- 0906 `dupl_get_or_put`:
property has already been specified
- 0907 `declspec_property_not_allowed`:
`__declspec(property)` is not allowed on this declaration
- 0908 `no_get_property`:
member is declared with `__declspec(property)`, but no "get" function was specified
- 0909 `get_property_function_missing`:
the `__declspec(property)` "get" function "xxxx" is missing
- 0910 `no_put_property`:
member is declared with `__declspec(property)`, but no "put" function was specified

- 0911 `put_property_function_missing`:
the `__declspec(property)` "put" function "xxxx" is missing
- 0912 `dual_lookup_ambiguous_name`:
ambiguous class member reference -- *entity-kind* "entity" (declared at line xxxx) used in preference to *entity-kind* "entity" (declared at line xxxx)
- 0913 `bad_allocate_segname`:
missing or invalid segment name in `__declspec(allocate("..."))`
- 0914 `declspec_allocate_not_allowed`:
`__declspec(allocate)` is not allowed on this declaration
- 0915 `dupl_allocate_segname`:
a segment name has already been specified
- 0916 `pm_virtual_base_from_derived_class`:
cannot convert pointer to member of derived class "*type*" to pointer to member of base class "*type*" -- base class is virtual
- 0917 `cl_invalid_instantiation_directory`:
invalid directory for instantiation files:
- 0918 `cl_one_instantiation_per_object_option_only_in_cplusplus`:
option "one_instantiation_per_object" can be used only when compiling C++
- 0919 `invalid_output_file`:
invalid output file: "xxxx"
- 0920 `cannot_open_output_file`:
cannot open output file: "xxxx"
- 0921 `cl_ii_file_name_incompatible_with_multiple_inputs`:
an instantiation information file name may not be specified when compiling several input files
- 0922 `cl_one_instantiation_per_object_incompatible_with_multiple_inputs`:
option "one_instantiation_per_object" may not be used when compiling several input files

- 0923 `cl_ambiguous_option`:
more than one command line option matches the abbreviation
"`--xxxx`";
- 0925 `cv_qualified_function_type`:
a type qualifier cannot be applied to a function type
- 0926 `cannot_open_definition_list_file`:
cannot open definition list file: "`xxxx`"
- 0927 `cl_late_tiebreaker_option_only_in_cplusplus`:
late/early tiebreaker option can be used only when compiling C++
- 0928 `cl_strict_ansi_incompatible_with_tsw_extensions`:
strict ANSI mode is incompatible with TASKING Embedded C++
extensions
- 0929 `tsw_embedded_extensions_not_allowed`:
TASKING Embedded C++ extensions not allowed
- 0930 `tsw_at_already_used`:
`_at()` can only be used once in a declaration
- 0931 `tsw_atbit_already_used`:
`_atbit()` can only be used once in a declaration
- 0932 `tsw_at_atbit_conflict`:
`_at()` and `_atbit()` cannot be used in the same declaration
- 0941 `tsw_expr_not_integral_or_fractional`:
expression must have integral or fractional type
- 0942 `tsw_expr_not_integral_or_enum_or_fractional`:
expression must have integral, enum or fractional type
- 0943 `cl_options_after_input_file_not_allowed`:
options are not allowed after the input file name
- 0944 `bad_va_start`:
incorrect use of `va_start`

- 0945 `bad_va_arg`:
incorrect use of `va_arg`
- 0946 `bad_va_end`:
incorrect use of `va_end`
- 0947 `cl_pending_instantiations_option_only_in_cplusplus`:
pending instantiations option can be used only when compiling C++
- 0948 `cl_invalid_import_directory`:
invalid directory for `#import` files:
- 0949 `cl_import_only_in_microsoft`:
an import directory can be specified only in Microsoft mode
- 0950 `ref_not_allowed_in_union`:
a member with reference type is not allowed in a union
- 0951 `typedef_not_allowed`:
"typedef" may not be specified here
- 0952 `redecl_changes_access`:
redeclaration of *entity-kind* "*entity*" alters its access
- 0953 `qualified_name_required`:
a class or namespace qualified name is required
- 0954 `implicit_int_on_main`:
return type "int" omitted in declaration of function "main"
- 0955 `invalid_inheritance_kind_for_class`:
pointer-to-member representation "`xxxx`" is too restrictive for *entity-kind* "*entity*"
- 0956 `implicit_return_from_non_void_function`:
missing return statement at end of non-void *entity-kind* "*entity*"
- 0957 `duplicate_using_decl`:
duplicate using-declaration of "*entity*" ignored

- 0958 `unsigned_enum_bit_field_with_signed_enumerator`:
enum bit-fields are always unsigned, but enum "*type*" includes negative enumerator
- 0959 `cl_class_name_injection_option_only_in_cplusplus`:
option "`class_name_injection`" can be used only when compiling C++
- 0960 `cl_arg_dependent_lookup_option_only_in_cplusplus`:
option "`arg_dep_lookup`" can be used only when compiling C++
- 0961 `cl_friend_injection_option_only_in_cplusplus`:
option "`friend_injection`" can be used only when compiling C++
- 0962 `invalid_name_after_template`:
name following "`template`" must be a member template
- 0964 `local_class_friend_requires_prior_decl`:
nonstandard local-class friend declaration — no prior declaration in the enclosing scope
- 0965 `nonstd_default_arg`:
specifying a default argument on this declaration is nonstandard
- 0966 `cl_nonstd_using_decl_option_only_in_cplusplus`:
option "`nonstd_using_decl`" can be used only when compiling C++
- 0967 `bad_return_type_on_main`:
return type of function "`main`" must be "`int`"
- 0968 `template_parameter_has_class_type`:
a template parameter may not have class type
- 0969 `default_arg_on_member_decl`:
a default template argument cannot be specified on the declaration of a member of a class template
- 0970 `return_from_ctor_function_try_block_handler`:
a return statement is not allowed in a handler of a function try block of a constructor

- 0971 `no_ordinary_and_extended_designators`:
ordinary and extended designators cannot be combined in an
initializer designation
- 0972 `no_negative_designator_range`:
the second subscript must not be smaller than the first
- 0973 `cl_designators_option_only_in_C`:
option "designators" can be used only when compiling C
- 0974 `cl_extended_designators_option_only_in_C`:
option "extended_designators" can be used only when compiling C
- 0975 `extra_bits_ignored`:
declared size for bit field is larger than the size of the bit field type;
truncated to `xxxx` bits
- 0976 `constructor_type_mismatch`:
type used as constructor name does not match type *"type"*
- 0977 `type_with_no_linkage_in_var_with_linkage`:
use of a type with no linkage to declare a variable with linkage
- 0978 `type_with_no_linkage_in_function`:
use of a type with no linkage to declare a function
- 0979 `return_type_on_constructor`:
return type may not be specified on a constructor
- 0980 `return_type_on_destructor`:
return type may not be specified on a destructor
- 0981 `malformed_universal_character`:
incorrectly formed universal character name
- 0982 `invalid_UCN`:
universal character name specifies an invalid character
- 0983 `UCN_names_basic_char`:
a universal character name cannot designate a character in the basic
character set

- 0984 `invalid_identifier_UCN`:
this universal character is not allowed in an identifier
- 0985 `VA_ARGS_not_allowed`:
the identifier `__VA_ARGS__` can only appear in the replacement lists of variadic macros
- 0986 `friend_qualification_ignored`:
the qualifier on this friend declaration is ignored
- 0987 `no_range_designator_with_dynamic_init`:
array range designators cannot be applied to dynamic initializers
- 0988 `property_name_not_allowed`:
property name cannot appear here
- 0989 `inline_qualifier_ignored`:
"inline" used as a function qualifier is ignored
- 0990 `cl_compound_literals_option_only_in_C`:
option "compound_literals" can be used only when compiling C
- 0991 `vla_not_allowed`:
a variable-length array type is not allowed
- 0992 `bad_integral_compound_literal`:
a compound literal is not allowed in an integral constant expression
- 0993 `bad_compound_literal_type`:
a compound literal of type "*type*" is not allowed
- 0994 `friend_template_in_local_class`:
a template friend declaration cannot be declared in a local class
- 0995 `ambiguous_question_operator`:
ambiguous "?" operation: second operand of type "*type*" can be converted to third operand type "*type*", and vice versa
- 0996 `bad_call_of_class_object`:
call of an object of a class type without appropriate `operator()` or conversion functions to pointer-to-function type

- 0997 surrogate_func_add_on:
surrogate function from conversion *name*
- 0998 ambiguous_class_call:
there is more than one way an object of type "*type*" can be called
for the argument list:
- 0999 expected_asm_before_endasm_pragma:
expected a pragma asm before pragma endasm
- 1000 end_of_source_reached_before_pragma_endasm:
end of source reached while searching for pragma endasm
- 1001 similar_typedef:
typedef name has already been declared (with similar type)
- 1002 no_internal_linkage_for_new_or_delete:
operator new and operator delete cannot be given internal linkage
- 1003 no_mutable_allowed_on_anonymous_union:
storage class "mutable" is not allowed for anonymous unions
- 1004 bad_pch_file:
invalid precompiled header file
- 1005 abstract_class_catch_type:
abstract class type "*type*" is not allowed as catch type:
- 1006 bad_qualified_function_type:
a qualified function type cannot be used to declare a nonmember
function or a static member function
- 1007 bad_qualified_function_type_parameter:
a qualified function type cannot be used to declare a parameter
- 1008 ptr_or_ref_to_qualified_function_type:
cannot create a pointer or reference to qualified function type
- 1009 nonstd_braces:
extra braces are nonstandard

- 1010 `bad_cmd_line_macro`:
invalid macro definition:
- 1011 `nonstandard_ptr_minus_ptr`:
subtraction of pointer types "*type*" and "*type*" is nonstandard
- 1012 `empty_template_param_list`:
an empty template parameter list is not allowed in a template
template parameter declaration
- 1013 `exp_class`:
expected "class"
- 1014 `struct_not_allowed`:
the "class" keyword must be used when declaring a template
template parameter
- 1015 `virtual_function_decl_hidden`:
entity-kind "*entity*" is hidden by "*entity*" -- virtual function override
intended?
- 1016 `no_qualified_friend_definition`:
a qualified name is not allowed for a friend declaration that is a
function definition
- 1017 `not_compatible_with_tmpl_tmpl_param`:
entity-kind "*entity*" is not compatible with *entity-kind* "*entity*"
- 1018 `storage_class_requires_function_or_variable`:
a storage class may not be specified here
- 1019 `member_using_must_be_visible_in_direct_base`:
class member designated by a using-declaration must be visible in a
direct base class
- 1020 `cl_sun_incompatible_with_microsoft`:
Sun mode is incompatible with Microsoft mode
- 1021 `cl_sun_incompatible_with_cfront`:
Sun mode is incompatible with cfront mode

- 1022 `cl_strict_ansi_incompatible_with_sun`:
strict ANSI mode is incompatible with Sun mode
- 1023 `cl_sun_mode_only_in_cplusplus`:
Sun mode is only allowed when compiling C++
- 1024 `template_template_param_same_name_as_tmpl_param`:
a template template parameter cannot have the same name as one of its template parameters
- 1025 `recursive_def_arg_instantiation`:
recursive instantiation of default argument
- 1026 `dependent_type_in_tmpl_tmpl_param`:
a parameter of a template template parameter cannot depend on the type of another template parameter
- 1027 `bad_template_name`:
entity-kind "entity" is not an entity that can be defined
- 1028 `destructor_name_must_be_qualified`:
destructor name must be qualified
- 1029 `no_typename_in_friend_class_decl`:
friend class name may not be introduced with "typename"
- 1030 `no_ctor_or_dtor_using_declaration`:
a using-declaration may not name a constructor or destructor
- 1031 `friend_is_nonreal_template`:
a qualified friend template declaration must refer to a specific previously declared template
- 1032 `bad_class_template_decl`:
invalid specifier in class template declaration
- 1033 `simple_incompatible_param`:
argument is incompatible with formal parameter

1034 `asmfunc_not_allowed:`
use 'extern "asm"' instead of '`_asmfunc`' for external assembly
functions

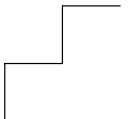
APPENDIX

B

UTILITY PROGRAMS



TASKING



B

APPENDIX

1 INTRODUCTION

This appendix describes the utility programs that are delivered with the C++ compiler. The utility programs help with various link-time issues and are meant to be called from the control program.



When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as '(', ')', and '?') must be enclosed with " " or escaped. The **-?** option (in the C-shell) becomes: **"-?"** or **-\\?**.

2 PRELINKER

The prelinker is invoked at link time by the control program to manage automatic instantiation of template entities. It is given a complete list of the object files and libraries that are to be linked together. It examines the external names defined and referenced within those files, and finds cases where template entities are referenced but not defined. It then examines information in the object files that describes instantiations that could have been done during compilation, and assigns the needed instantiations to appropriate files. The prelinker then invokes the compiler again to compile those files, which will do the necessary instantiations.

The invocation syntax of the C++ prelinker is:

prelk166 [*option*]... *files*

where the *files* list includes all object files and libraries, and the *options* are:

- ?** Display an explanation of options at **stdout**.
- V** Display version information at **stderr**.
- c c** Use *c* as symbol prefix character instead of the default underscore.
- D** Do not assign instantiation to non-local object files. Instantiations may only be assigned to object files in the current directory.
- i** Ignore invalid input lines.
- lxxx** Specify a library (e.g., **-lcp**).
- L** Skip system library search.

- L *directory*** Specify an additional search path for system libraries.
- m** Do not demangle identifier names that are displayed.
- n** Update the instantiation list files (*.ii*), but do not recompile the source files.
- N** If a file from a non-local directory needs to be recompiled, do the compilation in the current directory. An updated list of object files and library names is written to the file specified by the **-o** option so that the control program can tell that alternate versions of some of the object files should be used.
- o *file*** Write an updated list of object files and library names to the file specified by *file*. Use this option when the **-N** or **-O** option is used.
- O** One instantiation per object mode is used. A list of object files, including the instantiation object files associated with the object files specified on the prelinker command line, is written to the file specified by the **-o** option.
- q** Quiet mode. Turns off verbose mode.
- r** Do not stop after the maximum number of iterations. (The instantiation process is iterative: a recompilation may bring up new template entities that need to be instantiated, which requires another recompilation, etc. Some recursive templates can cause iteration that never terminates, because each iteration introduces another new entity that was not previously there. By default, this process is stopped after a certain number of iterations.)
- R *number*** Override the number of reserved instantiation information file lines to be used.
- s *number*** Specifies whether the prelinker should check for entities that are referenced as both explicit specializations and generated instantiations. If *number* is zero the check is disabled, otherwise the check is enabled.
- S** Suppress instantiation flags in the object files.
- T *cpu*** Set the target CPU type. This name is used to determine the actual location of the system libraries relative to the default **lib** directory.

- u** Specify that external names do not have an added leading underscore. By default, external names get a leading underscore. With this option you specify that the leading underscore belongs to the external name.
- v** Verbose mode.

3 MUNCHER

The muncher implements a lowest-common-denominator method for getting global initialization and termination code executed on systems that have no special support for that.

The muncher accepts the output of the linker as its input file and generates a C program that defines a data structure containing a list of pointers to the initialization and termination routines. This generated program is then compiled and linked in with the executable. The data structure is consulted at run-time by startup code invoked from `_main`, and the routines on the list are invoked at the appropriate times.

The invocation syntax of the C++ muncher is:

munch166 [*option*]... [*file*]

where the *file* is an output file generated by the linker, and the *options* are:

- ?** Display an explanation of options at `stdout`.
- V** Display version information at `stderr`.
- c c** Use *c* as symbol prefix character instead of the default underscore
- i n** Skip first *n* lines of input.
- o file** Write output to *file*.
- u** Specify that external names do not have an added leading underscore. By default, external names get a leading underscore. With this option you specify that the leading underscore belongs to the external name.

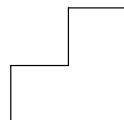


INDEX

INDEX



TASKING



INDEX

Symbols

#define, 3-31
 #include, 3-58, 3-113
 system include directory, 3-98
 #pragma, 3-116
 #undef, 3-103
 __ARRAY_OPERATORS, 2-36, 3-103
 __cplusplus, 2-36, 3-103
 __DATE__, 2-36, 3-103
 __EXCEPTIONS, 2-36
 __FILE__, 2-36, 3-103
 __LINE__, 2-36, 3-103
 __NAMESPACES, 2-37
 __PLACEMENT_DELETE, 2-37
 __RTTI, 2-37
 __SIGNED_CHARS__, 2-36, 3-93,
 3-103
 __STDC__, 2-36, 3-103
 __STDC_VERSION__, 2-36
 __TIME__, 2-36, 3-103
 __TSW_IMPLICIT_USING_STD, 2-37
 __TSW_RUNTIME_USES_NAMESPACES
 , 2-37
 __BOOL, 2-36, 3-103
 __MODEL, 2-36
 __WCHAR_T, 2-36, 3-103

A

alternative tokens, 3-18
 anachronism, 2-10
 anachronisms, 3-19, 3-26, 3-73
 ansi standard, 3-103
 array new and delete, 3-21
 automatic instantiation, 1-5
 automatic instantiation method, 2-29

B

bool keyword, 3-24

C

C++
 language extensions, 2-4, 2-7
 library, 2-3
 C++ dialect, 2-4, 2-7
 accepted, 2-7
 anachronisms accepted, 2-10
 cfront 2.1 and 3.0 extensions, 2-18
 cfront 2.1 extensions, 2-14
 new language features accepted, 2-7
 new language features not accepted,
 2-10
 normal C++ mode extensions, 2-12
 not accepted, 2-10
 C++ language features
 accepted, 2-7
 not accepted, 2-10
 c_plusplus, 2-36, 3-103
 can_instantiate, 3-116
 catastrophic error, 4-3
 cfront, 3-26
 2.1 and 3.0 extensions, 2-18
 2.1 extensions, 2-14
 character
 signed, 3-93
 unsigned, 3-105
 class name injection, 3-27
 command file, 3-47
 compiler diagnostics, 4-1
 compiler limits, 3-118
 compiler use, 3-1
 const, string literals, 3-29

copy assignment operator, 3-23
 CP166INC, 3-113
 cross-reference, 3-112

D

detailed option description, compiler,
 3-16-3-112
 development flow, 1-4
 diagnostics, 4-1
 brief, 3-25
 error severity, 3-33, 4-3
 TASKING style, 3-101
 wrap, 3-111
 digraph, 3-18
 directory separator, 3-114
 do_not_instantiate, 3-116
 dollar signs, 3-36

E

embedded C++, 3-38, 3-39
 entities, remove unneeded, 3-91
 enum overloading, 3-40
 environment variables
 A166INC, 1-8
 C166INC, 1-8
 CC166INC, 1-8
 CC166OPT, 1-8
 CP166INC, 1-8, 3-113
 LINK166, 1-8
 LM_LICENSE_FILE, 1-8
 LOCATE166, 1-8
 M166INC, 1-8
 overview of, 1-8
 PATH, 1-8
 TASKING_LIC_WAIT, 1-8
 TMPDIR, 1-8
 used by tool chain, 1-8

error, 4-3
 error level, 4-6
 error limit, 3-41
 error messages, A-1
 error number, 3-34
 error output file, 3-42
 error severity, 3-33, 4-3
 exception, 3-43
 exit status, 4-6
 explicit specifier, 3-44
 extension, 1-9
 .abs, 1-9
 .asm, 1-9
 .c, 1-9
 .cc, 1-9
 .cpp, 1-9
 .cxx, 1-9
 .hex, 1-9
 .ic, 1-9
 .lib, 1-9
 .lnl, 1-10
 .lno, 1-9
 .lst, 1-10
 .map, 1-10
 .mpe, 1-10
 .mpl, 1-10
 .obj, 1-9
 .out, 1-9
 .src, 1-9

extensions to C++, 2-4, 2-7
 extern C, 3-54
 extern C++, 3-54
 extern inline, 3-46

F

file extensions, 1-9, 3-3
 for-init statement, 3-49, 3-70
 friend injection, 3-51
 function names, unqualified, 3-20

G

guiding declarations, 3-53

H

hdrstop, 3-116

header stop, 2-38, 2-43

I

ident, 3-117

implicit inclusion, 2-35

include files, 3-113

at beginning of compilation, 3-59

default directory, 3-114

suffix, 3-57, 3-115

inline function, 3-46

inlining, 3-60

instantiate, 3-116

instantiation, 2-26

automatic, 2-29

directory, 3-63

one file per object, 3-82

pending, 3-88

template, 3-61

instantiation information file, 1-5

instantiation mode, 2-31

all, 2-32

local, 2-32

none, 2-31

used, 2-31

instantiation pragmas, 2-32

internal error, 4-3

introduction, 1-3

invocation, 3-3

K

keyword

bool, 3-24

typename, 3-102

wchar_t, 3-110

L

labels

__main, 2-46

_main, 2-46

language extensions, 3-95

language implementation, 2-1

library, 2-3

license, wait for available license, 1-8

license file, setting search directory,

1-8

lifetime, 3-66

limits, compiler, 3-118

list file, 3-64

long, arithmetic conversion rules, 3-67

lookup of unqualified function names,

3-20

M

macros

predefined, 2-36

variable argument list, 3-45, 3-108

main labels, 2-46

messages

diagnostic, 4-3

termination, 4-5

muncher, 1-7, B-5

N

namespace, 2-24, 3-69

std, 3-107

no_pch, 2-43, 3-116

O

once, 3-116

operator, keywords, 3-18

optimizations, c166, 2-46

option file, 3-47

options

-?, 3-17

#, 3-99

-\$, 3-36

--alternative-tokens, 3-18

--anachronisms, 3-19

--arg-dep-lookup, 3-20

--array-new-and-delete, 3-21

--auto-instantiation, 3-22

--base-assign-op-is-default, 3-23

--bool, 3-24

--brief-diagnostics, 3-25

--cfront-2.1, 3-26

--cfront-3.0, 3-26

--class-name-injection, 3-27

--comments, 3-28

--const-string-literals, 3-29

--create-pch, 3-30

--define, 3-31

--dependencies, 3-32

--diag-error, 3-33

--diag-remark, 3-33

--diag-suppress, 3-33

--diag-warning, 3-33

--display-error-number, 3-34

--distinct-template-signatures, 3-35

--dollar, 3-36

--early-tiebreaker, 3-37

--embedded, 3-38

--embedded-c++, 3-39

--enum-overloading, 3-40

--error-limit, 3-41

--error-output, 3-42

--exceptions, 3-43

--explicit, 3-44

--extended-variadic-macros, 3-45

--extern-inline, 3-46

--for-init-diff-warning, 3-49

--force-utbl, 3-50

--friend-injection, 3-51

--gen-c-file-name, 3-52

--guiding-decls, 3-53

--implicit-extern-c-type-conversion,
3-54

--implicit-include, 3-55

--implicit-typename, 3-56

--incl-suffixes, 3-57

--include-directory, 3-58

--include-file, 3-59

--inlining, 3-60

--instantiate, 3-61

--instantiation-dir, 3-63

--late-tiebreaker, 3-37

--list-file, 3-64

--long-lifetime-temps, 3-66

--long-preserving-rules, 3-67

--namespaces, 3-69

--new-for-init, 3-70

--no-alternative-tokens, 3-18

--no-anachronisms, 3-19

--no-arg-dep-lookup, 3-20

--no-array-new-and-delete, 3-21

--no-auto-instantiation, 3-22

--no-base-assign-op-is-default,
3-23

--no-bool, 3-24

--no-brief-diagnostics, 3-25

--no-class-name-injection, 3-27

--no-code-gen, 3-71

--no-const-string-literals, 3-29

- no-distinct-template-signatures*, 3-35
- no-embedded*, 3-38
- no-enum-overloading*, 3-40
- no-exceptions*, 3-43
- no-explicit*, 3-44
- no-extended-variadic-macros*, 3-45
- no-extern-inline*, 3-46
- no-for-init-diff-warning*, 3-49
- no-friend-injection*, 3-51
- no-guiding-decls*, 3-53
- no-implicit-extern-c-type-conversion*, 3-54
- no-implicit-include*, 3-55
- no-implicit-typename*, 3-56
- no-inlining*, 3-60
- no-line-commands*, 3-72
- no-long-preserving-rules*, 3-67
- no-namespaces*, 3-69
- no-nonconst-ref-anachronism*, 3-73
- no-nonstd-qualifier-deduction*, 3-74
- no-nonstd-using-decl*, 3-75
- no-old-specializations*, 3-80
- no-preproc-only*, 3-76
- no-remove-unneeded-entities*, 3-91
- no-rtti*, 3-92
- no-special-subscript-cost*, 3-94
- no-tsw-diagnostics*, 3-101
- no-typename*, 3-102
- no-use-before-set-warnings*, 3-77
- no-using-std*, 3-107
- no-variadic-macros*, 3-108
- no-warnings*, 3-78
- no-wchar_t-keyword*, 3-110
- no-wrap-diagnostics*, 3-111
- nonconst-ref-anachronism*, 3-73
- nonstd-qualifier-deduction*, 3-74
- nonstd-using-decl*, 3-75
- old-for-init*, 3-70
- old-line-commands*, 3-79
- old-specializations*, 3-80
- old-style-preprocessing*, 3-81
- one-instantiation-per-object*, 3-82
- output*, 3-83
- pch*, 3-84
- pch-dir*, 3-85
- pch-messages*, 3-86
- pch-verbose*, 3-87
- pending-instantiations*, 3-88
- preprocess*, 3-89
- remarks*, 3-90
- remove-unneeded-entities*, 3-91
- rtti*, 3-92
- short-lifetime-temps*, 3-66
- signed-chars*, 3-93
- special-subscript-cost*, 3-94
- strict*, 3-95
- strict-warnings*, 3-95
- suppress-typeinfo-vars*, 3-96
- suppress-utbl*, 3-97
- sys-include*, 3-98
- timing*, 3-99
- trace-includes*, 3-100
- tsw-diagnostics*, 3-101
- typename*, 3-102
- undefine*, 3-103
- unsigned-chars*, 3-105
- use-pch*, 3-106
- using-std*, 3-107
- variadic-macros*, 3-108
- version*, 3-109
- wchar_t-keyword*, 3-110
- wrap-diagnostics*, 3-111
- xref*, 3-112
- A, 3-95
- a, 3-95
- B, 3-55
- b, 3-26
- C, 3-28
- D, 3-31
- E, 3-89
- e, 3-41

-f, 3-47
 -H, 3-100
 -I, 3-58
 -j, 3-77
 -L, 3-64
 -M, 3-32
 -Mmodel, 3-68
 -n, 3-71
 -o, 3-52
 -P, 3-72
 -r, 3-90
 -s, 3-93
 -T, 3-22
 -t, 3-61
 -U, 3-103
 -u, 3-105
 -V, 3-109
 -v, 3-109
 -w, 3-78
 -X, 3-112
 -x, 3-43
 detailed description, 3-16
 overview, 3-3
 overview in functional order, 3-10
 priority, 3-3
 output file, 3-52, 3-83
 overview, 1-1

P

pch mode
 automatic, 2-38, 3-84
 manual, 2-42, 3-30, 3-106
 pragma
 can_instantiate, 2-33, 3-116
 do_not_instantiate, 2-32, 3-116
 hdrstop, 2-38, 2-43, 3-116
 ident, 3-117
 instantiate, 2-32, 3-116
 no_pch, 2-43, 3-116
 once, 3-116

pragmas, 3-116
 precompiled header, 2-38
 automatic, 2-38, 3-84
 create, 2-42, 3-30
 directory, 2-42, 2-43, 3-85
 file cannot be used, 3-87
 manual, 2-42
 messages, 3-86
 performance, 2-44
 pragmas, 2-43
 prefix, 2-41
 use, 2-42, 3-106
 predefined macros, 2-36
 predefined symbols, 3-103
 prelinker, 1-5, B-3
 prelinker prelk166, 2-29

Q

qualifier deduction, 3-74

R

raw listing, 3-64
 remark, 4-3
 remarks, 3-90
 return values, 4-6
 run-time type information, 3-92

S

signals, 4-6
 stack, 2-26
 STLport library, 2-3
 string literals, *const*, 3-29
 suffix, *include file*, 3-115
 symbols, *predefined*, 3-103
 syntax checking, 3-71

system include directory, 3-98, 3-115

typename keyword, 3-102

T

template, 2-26

distinct signatures, 3-35

guiding declarations, 3-53

specialization, 3-80

template instantiation, 2-26

#pragma directives, 2-32

automatic, 2-27, 3-22

directory, 3-63

implicit inclusion, 2-35, 3-55

instantiation modes, 2-31, 3-61

one file per object, 3-82

pending, 3-88

temporary files, setting directory, 1-8

tie-breakers, 3-37

timing information, 3-99

tool chain, 1-4

muncher, 1-7

prelinker, 1-5

type information, 3-96

U

using declaration, allow unqualified

 name, 3-75

utilities, B-1

muncher, B-5

prelinker, B-3

V

version information, 3-109

virtual function table, 3-50, 3-97

W

warning, 4-3

warnings (suppress), 3-77, 3-78

wchar_t keyword, 3-110

