

C++ Front End

Internal Documentation (excerpt)

December 21, 2011
(Version 4.4)

Copyright © 1992-2011 Edison Design Group, Inc.

11 Rocky Way, West Orange NJ 07052 973-325-6840

Chapter 1

External Interface

1

1.1	Command Line	1
1.1.1	Language Dialect Options	2
1.1.2	Input/output Options	4
1.1.3	Preprocessor Options	6
1.1.4	Diagnostic Options	8
1.1.5	Individual Language Features	10
1.1.6	Alternative Language Behaviors	14
1.1.7	Template Instantiation Options	19
1.1.8	Code Generation Options	21
1.1.9	Precompiled Header Options	22
1.1.10	Front End Debugging Options	23
1.1.11	Miscellaneous Options	24
1.2	Environment Variables	24
1.3	Diagnostic Messages	24
1.4	Termination Messages	26
1.5	Response to Signals	27
1.6	Exit Status	27
1.7	Finding Include Files	27
1.8	C++ Dialect Accepted	28
1.8.1	Standard Language Features Accepted	28
1.8.2	C++0x Language Features Accepted	31
1.8.3	Anachronisms Accepted	33
1.8.4	Extensions Accepted in Normal C++ Mode	34
1.8.5	Extensions Accepted in Cfront 2.1 Compatibility Mode	36
1.8.6	Extensions Accepted in Cfront 2.1 and 3.0 Compatibility Mode	39
1.8.7	Extensions Accepted in Sun Compatibility Mode	43
1.9	C Dialect Accepted	45
1.9.1	C99 Features Available in Other Modes	45
1.9.2	ANSI C Extensions	47
1.9.3	K&R/pcc Mode	51
1.9.4	Extensions Accepted in SVR4 Compatibility Mode	55
1.9.5	Microsoft Mode	55
1.10	GNU Extensions	84
1.10.1	Extensions Only Accepted in GNU C Mode	87
1.10.2	Extensions Only Accepted in GNU C++ Mode	89
1.11	Namespace Support	98
1.12	Template Instantiation	100
1.12.1	Automatic Instantiation	101
1.12.2	Instantiation Modes	103
1.12.3	Instantiation #pragma Directives	104
1.12.4	Implicit Inclusion	105
1.12.5	Exported Templates	106
1.13	Extern Inline Functions	109

1.14	Predefined Macros	109
1.14.1	Pragmas	113
1.14.2	Instantiation Pragmas	113
1.14.3	Precompiled Header Pragmas	113
1.14.4	Once Pragma	113
1.14.5	Diagnostic Pragmas	114
1.14.6	Pack Pragma	114
1.14.7	Microsoft Pragmas	116
1.15	Precompiled Headers	116
1.15.1	Automatic Precompiled Header Processing	116
1.15.2	Manual Precompiled Header Processing	119
1.15.3	Other Ways for Users to Control Precompiled Headers	119
1.15.4	Performance Issues	120

Chapter 1

External Interface

Bear in mind that this information applies to the standard product. If any local modifications are made, the documentation given to users must be adjusted accordingly.

1.1 Command Line

The front end is invoked by a command of the form

```
edgcpfe [options] ifile
```

to compile the single input file *ifile*. If - (hyphen) is specified for *ifile*, the front end reads from `stdin`.¹ No particular file name suffix is required on input files.

Command line options may be specified using either single character option codes (e.g., -o) or keyword options (e.g., --output). A single character option specification consists of a hyphen followed by one or more option characters (e.g., -Ab). 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). 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.

If the configuration flag `COMPILE_MULTIPLE_SOURCE_FILES` is defined as `TRUE`, a list of files may appear for *ifile*. If a list of files is specified, options that specify a compilation output file (--output, --list, and --xref) may not be used, and the name of each source file is written to the error output file as the compilation of that file begins.

When one of the preprocessing-only modes is specified (see below), the --output option can be used to specify the preprocessing output file. If --output is not specified, preprocessing output is written to `stdout`. Preprocessing output has trigraphs and line splices processed (and thus they do not appear in their original form).

¹ This is not recommended in general, since diagnostic messages and the like will then not include a file name or will refer to the file name “-”.

When compilation (rather than just preprocessing) is done, the output (if any) from the compilation is written to a file selected by the back end; see the documentation of the back end for further information. For versions of the front end that generate an intermediate language file, the `--output` option can be used to specify the IL output file.

The *options* are described in the following subsections (organized by category).

1.1.1 Language Dialect Options

The following options select among various C and C++ dialects.

<code>--anachronisms</code>	
<code>--no_anachronisms</code>	Enable or disable anachronisms in C++ mode. The default for this option is specified by <code>DEFAULT_ALLOW_ANACHRONISMS</code> . This option is valid only in C++ mode.
<code>--c</code>	
<code>-m</code>	Enable compilation of C rather than C++.
<code>--c89</code>	Enable the compilation of the C89 version of C.
<code>--c99</code>	
<code>--no_c99</code>	Enable or disable compilation of the C99 version of C. In either case, C rather than C++ is compiled.
<code>--old_c</code>	
<code>-K</code>	Enable K&R/pcc mode, which approximates the behavior of the standard UNIX <code>pcc</code> . ANSI/ISO C features that do not conflict with K&R/pcc features are still supported in this mode.
<code>--c++</code>	
<code>-p</code>	Enable compilation of C++. This is the default.
<code>--c++11</code>	
<code>--no_c++11</code>	
<code>--c++0x</code>	
<code>--no_c++0x</code>	Enable or disable extensions added in the C++11 standard. This option is only valid in C++ mode and can be combined with the options enabling strict ANSI/ISO conformance.
<code>--cfront_2.1</code>	
<code>-b</code>	Enable compilation of C++ with compatibility with cfront version 2.1. 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. This option also enables acceptance of anachronisms.
<code>--cfront_3.0</code>	Enable compilation of C++ with compatibility with cfront version 3.0. 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 3.0. This option also enables acceptance of anachronisms.
<code>--c++cli</code>	
<code>--cppcli</code>	
<code>--clr</code>	Enable compilation of C++/CLI source. This option implies Microsoft C++ mode; it cannot be combined with options that are contrary to such a mode (e.g.,

1.1 Command Line

forcing strict mode) or with the option `--microsoft_version` with a value less than 1600.

`--embedded_c`
`--no_embedded_c`

Enable or disable support for all Embedded C language extensions (fixed-point types, named address space qualifiers, and named-register storage class specifiers). These options are not available with C++ modes.

`--embedded_c++`

Enable the diagnosis 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).

`--g++`
`--no_g++`

Enable or disable GNU C++ language compatibility features. This option also specifies that the source language being compiled is C++. It is only recognized when `GNU_EXTENSIONS_ALLOWED` is TRUE.

`--gcc`
`--no_gcc`

Enable or disable GNU C language compatibility features. This option also specifies that the source language being compiled is C. It is only recognized when `GNU_EXTENSIONS_ALLOWED` is TRUE.

`--gnu_version` *version-number*

The version of the GNU compiler that should be emulated in any of the GNU (C or C++) modes. Version *x.y.z* of the GNU compiler is encoded as $x*10000+y*100+z$ (e.g., version 3.4.1 is emulated with the option “`--gnu_version 30401`”).

`--microsoft`
`--microsoft_16`
`--no_microsoft`

Enable or disable recognition of Microsoft extensions. These options may only be used if `MICROSOFT_EXTENSIONS_ALLOWED` is TRUE. `--microsoft` enables 32-bit mode. `--microsoft_16` enables 16-bit mode (i.e., support for `near` and `far`; the option is accepted only if `NEAR_AND_FAR_ALLOWED` is TRUE). The default values are supplied by `DEFAULT_MICROSOFT_MODE` and `DEFAULT_NEAR_AND_FAR_ENABLED`. When Microsoft extensions are recognized, language features that are not recognized by the Microsoft compiler are disabled by default. In most cases these features can then be enabled through use of other command line options (for example, `--bool`). Microsoft bugs mode is also enabled when `DEFAULT_MICROSOFT_BUGS` is TRUE.

`--microsoft_version` *version-number*

The version of the Microsoft compiler that should be emulated in Microsoft mode. This enables or disables particular Microsoft mode features when the acceptance of those features varies between versions of the Microsoft compiler. The value specified is the value of the predefined macro `_MSC_VER` supplied by the version of the Microsoft compiler to be emulated. For example, 1100 is the value that corresponds to Visual C++ version 5.0. This option also enables Microsoft mode, but it does not imply either 16-bit or 32-bit mode (i.e., it uses whatever value has already been specified, or the default value if none has been explicitly specified).

<code>--microsoft_bugs</code>	
<code>--no_microsoft_bugs</code>	Enable or disable recognition of certain Microsoft bugs. These options also enable Microsoft mode. Microsoft bugs mode is automatically enabled when Microsoft mode is used and <code>DEFAULT_MICROSOFT_BUGS</code> is <code>TRUE</code> .
<code>--nonstd_gnu_keywords</code>	
<code>--no_nonstd_gnu_keywords</code>	In GNU modes, <code>--no_nonstd_gnu_keywords</code> disables GNU-specific keywords that do not start with underscores (notably: <code>typeof</code>); the option has no effect in other modes. <code>--nonstd_gnu_keywords</code> is only accepted in GNU modes and enables any keywords that were disabled by a prior occurrence of <code>--no_nonstd_gnu_keywords</code> .
<code>--strict_warnings</code> or <code>-a</code>	
<code>--strict</code> or <code>-A</code>	Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C or C++. This is compatible with both C and C++ mode. It is not compatible with <code>pcc</code> mode. ANSI/ISO violations can be issued as either warnings or errors depending on which command line option is used. The <code>-A</code> and <code>--strict</code> options cause errors to be issued whereas the <code>-a</code> and <code>--strict_warnings</code> options produce warnings. The error threshold is set so that the requested diagnostics will be listed.
<code>--sun</code>	
<code>--no_sun</code>	Enable or disable Sun CC (version 5.0) language compatibility features. This option can only be used in C++ mode, and cannot be combined with options to enable strict ANSI mode, or compatibility with <code>cfront</code> , Microsoft, or GNU. It is only recognized when <code>SUN_EXTENSIONS_ALLOWED</code> is <code>TRUE</code> .
<code>--svr4</code>	
<code>--no_svr4</code>	Enable or disable recognition of SVR4 C compatibility features. The default value is supplied by <code>DEFAULT_SVR4_C_MODE</code> . This option also specifies that the source language being compiled is ANSI C.
<code>--upc</code>	
<code>--no_upc</code>	Enable or disable UPC (Unified Parallel C) extensions.
<code>--upc_threads n</code>	Specify a fixed number of UPC threads (this is the value of the <code>THREADS</code> identifier).
<code>--upc_strict</code>	
<code>--upc_relaxed</code>	Specify the default UPC access method for objects of shared types.

1.1.2 Input/output Options

The following options control the kind of input read or output produced or the location of such input or output.

<code>--unicode_source_kind kind</code>	Source input files that do not begin with a byte order mark indicating the kind of Unicode encoding are assumed to use the Unicode encoding indicated by <i>kind</i> . Possible values of <i>kind</i> are <code>UTF-8</code> , <code>UTF-16</code> , <code>UTF-16LE</code> (little-endian), <code>UTF-16BE</code> (big-endian), and <code>none</code> (to indicate that the source file is not Unicode).
---	--

1.1 Command Line

- `--gen_c_file_name` *file-name* When using the C-generating back end or the C++-generating back end, this option specifies the file name to be used for the generated output.
- `--dependencies`
`-M` Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of dependency lines suitable for input to the UNIX make program. Note that when implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies unless `--no_preproc_only` is also used.
- `--error_output` *efile* Redirect the output that would normally go to the error output file (i.e., 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.
- `--list` *lfile*
`-Llfile` Generate raw listing information in the file *lfile*. 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 front end. 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). Comments are replaced by a single space in the expanded-form line.
- 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.

<code>--list_macros</code>	Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of all the macros in effect after processing the source file, including predefined and command-line macros along with their definitions.
<code>--no_line_commands</code> <code>-P</code>	Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and without line control information.
<code>--old_line_commands</code>	When generating source output (e.g., with the C-generating back end), put out <code>#line</code> directives in the form used by the Reiser <code>cpp</code> , i.e., " <code># nnn</code> " instead of " <code>#line nnn</code> ".
<code>--output ofile</code> <code>-o ofile</code>	Specify the output file of the compilation, i.e., the preprocessing or intermediate language output file.
<code>--xref xfile</code> <code>-Xxfile</code>	Generate cross-reference information in the file <i>xfile</i> . For each reference to an identifier in the source program, a line of the form <i>symbol-id name ref-code file-name line-number column-number</i> is written, where <i>ref-code</i> is D for definition, d for declaration (that is, a declaration that is not a definition), T for a full template instantiation, t for a partial template instantiation (i.e., an instantiation of the template declaration, but not of its 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. <i>symbol-id</i> is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

1.1.3 Preprocessor Options

The following options control the behavior of the preprocessor.

<code>--comments</code> <code>-C</code>	Keep comments in the preprocessed output. This should be specified after either <code>--preprocess</code> or <code>--no_line_commands</code> ; it does not of itself request preprocessing output.
<code>--define_macro name [(parm-list)] [= def]</code> <code>-D name [(parm-list)] [= def]</code>	

1.1 Command Line

	Define macro <i>name</i> as <i>def</i> . If “= <i>def</i> ” is omitted, define <i>name</i> as 1. Function-style macros can be defined by appending a macro parameter list to <i>name</i> .
<code>--import_dir <i>dir-name</i></code>	Specify the directory in which the files to be included by <code>#import</code> directives are to be found. These files must have been generated previously by the Microsoft compiler. This option is valid only in Microsoft mode.
<code>--incl_suffixes <i>str</i></code>	Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. The argument is a colon-separated list of suffixes (e.g., “h:hpp:”). If a null suffix is to be allowed, it must be included in the suffix list. The default is specified by the configuration flag <code>DEFAULT_INCLUDE_FILE_SUFFIX_LIST</code> .
<code>--include_directory <i>dir</i></code> <code>--sys_include <i>dir</i></code> <code>-I<i>dir</i></code>	Add <i>dir</i> to the list of directories searched for <code>#includes</code> . See section 1.7, “Finding Include Files,” on page 28.
<code>--old_style_preprocessing</code>	Forces pcc style preprocessing when compiling in ANSI C or C++ mode. This may be used when compiling an ANSI C or C++ program on a system in which the system header files require pcc style preprocessing.
<code>--preinclude_macros <i>filename</i></code> <code>--preinclude <i>filename</i></code>	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 file name is searched for in the directories on the include search list. When the <code>preinclude_macros</code> variant is used, only the preprocessing directives from the file are evaluated. All of the actual code is discarded. The effect of this option is that any macro definitions from the specified file will be in effect when the primary source file is compiled. All of the macro-only files are processed before any of the normal preincludes. Within each group, the files are processed in the order in which they were specified.
<code>--preprocess</code> <code>-E</code>	Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and with line control information.
<code>--no_preproc_only</code>	May be used in conjunction with the options that normally cause the front end to do preprocessing only (e.g., <code>--preprocess</code> , 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.
<code>--stdc_zero_in_system_headers</code> <code>--no_std_c_zero_in_system_headers</code>	Enable or disable special processing of the <code>__STDC__</code> macro when referenced from system headers. When this feature is enabled, <code>__STDC__</code> has the value zero in system header files (files found in directories specified using the <code>--sys_include</code> option) and one otherwise. This feature is provided to emulate the behavior of gcc in certain configurations. The default is specified by <code>DEFAULT_STDC_ZERO_IN_SYSTEM_HEADERS</code> . If the flag is not TRUE by default, the

GNU mode default is specified by the `DEFAULT_GNU_STDC_ZERO_IN_SYSTEM_HEADERS` macro.

`--trace_includes`
`-H` Output a list of the names of files `#included` to the error output file. The source file is compiled normally (i.e., it is not just preprocessed) unless another option that causes preprocessing only is specified.

`--undefine_macro name`
`-Uname` Remove any initial definition of the macro *name*. `--undefine_macro` options are processed after all `--define_macro` options in the command line have been processed.

`--check_concatenations`
`--no_check_concatenations` Specifies whether or not the front end will issue a diagnostic if a concatenation operator `##` in a macro expansion fails to produce a valid preprocessor token as required by the C and C++ Standards.

`--no_token_separators_in_pp_output` Used in connection with options that generate preprocessing output, suppresses the extra spaces the front end inserts in the text of macro expansions to ensure that tokens that should not be combined remain separate tokens. This can be used to prevent the insertion of extraneous spaces when the front end is used as a preprocessor for text that is not C or C++.

1.1.4 Diagnostic Options

The following options control when or how diagnostics are emitted.

`--brief_diagnostics`
`--no_brief_diagnostics` 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.

`--context_limit number` Set the context limit to *number*. The context limit is the maximum number of template instantiation context entries to be displayed as part of a diagnostic message. The default is specified by the `DEFAULT_CONTEXT_LIMIT` macro. If the number of context entries exceeds the limit, the first and last *N* context entries are displayed, where *N* is half of the context limit. A value of zero is used to indicate that there is no limit.

`--diag_suppress tag, tag,...`
`--diag_remark tag, tag,...`
`--diag_warning tag, tag,...`
`--diag_error tag, tag,...` 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.

`--diag_once tag, tag,...` Causes the specified diagnostic to be issued only once as a warning or remark. The message(s) may be specified using a mnemonic error tag or using an error

1.1 Command Line

number. The error tag names and error numbers are listed in the Error Messages appendix.

`--display_error_number`

`--no_display_error_number`

Enable or disable the display of an 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 configuration macro `DEFAULT_DISPLAY_ERROR_NUMBER` specifies the default behavior.

`--error_limit number`

`-e number`

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

`--for_init_diff_warning`

`--no_for_init_diff_warning`

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. `DEFAULT_WARNING_ON_FOR_INIT_DIFFERENCE` specifies the default behavior. This option is valid only in C++ mode.

`--macro_positions_in_diagnostics`

`--no_macro_positions_in_diagnostics`

Enable or disable display of extra information for diagnostics that refer to text in macro expansions. These options may only be used if `FULLY_RESOLVED_MACRO_POSITIONS` is `TRUE`. The additional information includes the original source line from which the text identified by the message was copied into the macro expansion (i.e., a macro definition or an argument in the top-level macro invocation), as well as the chain of macro invocations that led to the expanded text designated by the diagnostic (if `MACRO_INVOCATION_TREE_IN_IL` is `TRUE`). The default value is given by `DEFAULT_MACRO_POSITIONS_IN_DIAGNOSTICS`.

`--remarks`

`-r`

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

`--report_gnu_extensions`

Warn about the use of certain GNU extensions outside system header files. Only valid in GNU C or C++ mode.

`--template_typedefs_in_diagnostics`

`--no_template_typedefs_in_diagnostics`

Enable or disable the replacement of typedefs declared in template classes with their underlying type. Diagnostic messages are often more useful when such typedefs are replaced. The default behavior is specified by the `DEFAULT_TEMPLATE_TYPEDEFES_IN_DIAGNOSTICS` macro.

`--timing`
`-#` 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.

`--no_use_before_set_warnings`
`-j` Suppress warnings on local automatic variables that are used before their values are set. The front end's algorithm for detecting such uses is conservative and is likely to miss some cases that an optimizer with sophisticated flow analysis could detect; thus, an implementation might choose to suppress the warnings from the front end when optimization has been requested but to permit them when the optimizer is not being run.

`--no_warnings`
`-w` Suppress warnings. Errors are still issued.

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

1.1.5 Individual Language Features

The following options enable or disable specific language syntax. (Options that affect the meaning of existing constructs are listed in Section 1.1.6 on page 14.)

`--alternative_tokens`
`--no_alternative_tokens` Enable or disable recognition of alternative tokens. This controls recognition of the digraph tokens in C and C++, and controls recognition of the operator keywords (e.g., `and`, `bitand`, etc.) in C++. The default behavior is specified by the `DEFAULT_ALTERNATIVE_TOKENS_ALLOWED` configuration constant.

`--array_new_and_delete`
`--no_array_new_and_delete` Enable or disable support for array `new` and `delete`. The default behavior is specified by the configuration constant `DEFAULT_ARRAY_NEW_AND_DELETE_ENABLED`. This option is valid only in C++ mode. The front end can be configured to define a preprocessing variable when array `new` and `delete` are enabled. This preprocessing variable may then be used by the standard header files to decide whether to include the declarations for the array `new/delete` functions.

`--auto_storage`
`--no_auto_storage` Enable or disable the traditional meaning of `auto` as a storage class specifier (only valid in C++ mode). (If disabled, then `auto` must be a type specifier.) The default behavior is specified by the `DEFAULT_AUTO_STORAGE_CLASS_SPECIFIER_ENABLED` configuration constant.

`--auto_type`
`--no_auto_type` Enable or disable `auto` as a type specifier where the actual type is deduced from an initializer (only valid in C++ mode). This is a C++11 feature. If enabled, the traditional meaning of `auto` as a storage class specifier is disabled (which matches C++11), unless the option `--auto_storage` is also supplied. The

1.1 Command Line

default behavior is specified by the `DEFAULT_AUTO_TYPE_SPECIFIER_ENABLED` configuration constant.

- `--bool`
`--no_bool` Enable or disable recognition of `bool`. The default value is supplied by `DEFAULT_BOOL_IS_KEYWORD`. This option is valid only in C++ mode. The front end can be configured to define a preprocessing variable when `bool` is recognized. This preprocessing variable may then be used by header files to determine whether a typedef should be supplied to define `bool`.
- `--c++11_sfinae`
`--no_c++11_sfinae` Enable or disable template deduction in the style dictated by the C++11 standard, i.e., where general expressions are allowed in deduction contexts and they undergo the full usual semantic checking. This type of deduction is necessary to get the full power of the `decltype` feature in return types. (“SFINAE” refers to the initials of the phrase “Substitution Failure Is Not An Error”, which is the guiding principle for template deduction, and by extension a name for the process of deduction.)
- `--c++11_sfinae_ignore_access`
`--no_c++11_sfinae_ignore_access` When new-style SFINAE is enabled, these options control whether or not access errors cause deduction failures. The positive option indicates that access checking should not be done during template deduction, and therefore that access errors cannot cause a deduction failure; this was the status quo ante when the new-style SFINAE rules were proposed. The “no” version of the option indicates that access checking should be done during template deduction, and therefore that access errors cause a deduction failure; this is the revised direction chosen by the standards committee.
- `--compound_literals`
`--no_compound_literals` Enable or disable support for “compound literals” (a C99 feature).
- `--designators`
`--no_designators` Enable or disable support for designators (a C99 feature). These options are not available in C++ mode.
- `--extended_designators`
`--no_extended_designators` Enable or disable support for “extended designators,” an extension accepted only in C mode to emulate the behavior of certain other C compilers when it comes to designators in aggregate initializers.
- `--dollar`
`-$` Accept dollar signs in identifiers. The default value of this option is specified in a configuration file.
- `--exceptions`
`--no_exceptions`
`-x` Enable or disable support for exception handling. `-x` is equivalent to `--exceptions`. The default behavior is specified by the configuration constant `DEFAULT_EXCEPTIONS_ENABLED`. This option is valid only in C++ mode.

<code>--explicit</code> <code>--no_explicit</code>	Enable or disable support for the <code>explicit</code> specifier on constructor declarations. The default behavior is specified by the configuration constant <code>DEFAULT_EXPLICIT_KEYWORD_ENABLED</code> . This option is valid only in C++ mode.
<code>--export</code> <code>--no_export</code>	Enable or disable recognition of exported templates. The default value is supplied by <code>DEFAULT_EXPORT_TEMPLATE_ALLOWED</code> . This option is valid only in C++ mode. This option requires that dependent name processing be done, and cannot be used with implicit inclusion of template definitions.
<code>--extern_inline</code> <code>--no_extern_inline</code>	Enable or disable support for <code>inline</code> functions with external linkage in C++. When <code>inline</code> functions are allowed to have external linkage (as required by the standard), then <code>extern</code> and <code>inline</code> are compatible specifiers on a nonmember function declaration; the default linkage when <code>inline</code> appears alone is external (that is, <code>inline</code> means <code>extern inline</code> on nonmember functions); and an <code>inline</code> member function takes on the linkage of its class (which is usually external). However, when <code>inline</code> functions have only internal linkage (as specified in the ARM), then <code>extern</code> and <code>inline</code> are incompatible; the default linkage when <code>inline</code> appears alone is internal (that is, <code>inline</code> means <code>static inline</code> on nonmember functions); and <code>inline</code> member functions have internal linkage no matter what the linkage of their class.
<code>--fixed_point</code> <code>--no_fixed_point</code>	Enable or disable support for Embedded C fixed-point types. (Not available in C++ modes.)
<code>--lambdas</code> <code>--no_lambdas</code>	Enable or disable support for C++11 lambdas. (Only available in C++ modes.)
<code>--long_long</code>	Permit the use of <code>long long</code> in strict mode in dialects in which it is non-standard.
<code>--multibyte_chars</code> <code>--no_multibyte_chars</code>	Enable or disable processing for multibyte character sequences in comments, string literals, and character constants. The default behavior is specified by <code>DEFAULT_MULTIBYTE_CHARS_IN_SOURCE_ALLOWED</code> . Multibyte encodings are used for character sets like the Japanese SJIS.
<code>--named_address_spaces</code> <code>--no_named_address_spaces</code>	Enable or disable support for Embedded C named address space qualifiers. (Not available in C++ modes.)
<code>--named_registers</code> <code>--no_named_registers</code>	Enable or disable support for Embedded C named-register storage class specifiers. (Not available in C++ modes.)
<code>--namespaces</code> <code>--no_namespaces</code>	Enable or disable support for namespaces. The default behavior is specified by the configuration constant <code>DEFAULT_NAMESPACES_ENABLED</code> . This option is valid only in C++ mode.

1.1 Command Line

`--nonstd_using_decl`
`--no_nonstd_using_decl` In C++, controls whether a nonmember using-declaration that specifies an unqualified name is allowed.

`--nullptr`
`--no nullptr` Enable or disable support for the C++11 `nullptr` keyword. (Only valid in C++ mode.)

`--old_specializations`
`--no_old_specializations` Enable or disable acceptance of old-style template specializations (i.e., specializations that do not use the `template<>` syntax). Default behavior is specified by `DEFAULT_OLD_SPECIALIZATIONS_ALLOWED`. This option is valid only in C++ mode.

`--restrict`
`--no_restrict` Enable or disable recognition of the `restrict` keyword.

`--rtti`
`--no_rtti` Enable or disable support for RTTI (runtime type information) features: `dynamic_cast`, `typeid`. The default behavior is specified by the configuration constant `DEFAULT_RTTI_ENABLED`. This option is valid only in C++ mode.

`--rvalue_ctor_is_copy_ctor`
`--rvalue_ctor_is_not_copy_ctor` Determines whether an rvalue (or “move”) constructor is treated as a copy constructor (the default) or not. If rvalue constructors are treated as copy constructors, a user-declared rvalue constructor will inhibit the implicit generation of a traditional copy constructor.

`--rvalue_refs`
`--no_rvalue_refs` Enable or disable support for rvalue references. (Only valid in C++ mode.)

`--sun_linker_scope`
`--no_sun_linker_scope` Enable or disable support for Sun CC 5.5 link scope specifiers (`__global`, `__symbolic`, and `__hidden`). The default setting is supplied by `DEFAULT_SUN_LINKER_SCOPE_ALLOWED`.

`--thread_local_storage`
`--no_thread_local_storage` Enable or disable the `__thread` specifier to indicate that variables should be stored in thread-local storage. This option is enabled by default in Sun mode. It is also enabled by default in GNU mode when `gnu_version` \geq 30400.

`--trigraphs`
`--no_trigraphs` Enable or disable recognition of trigraphs. The default behavior is specified by the `DEFAULT_TRIGRAPHS_ALLOWED` configuration constant.

`--typename`
`--no_typename` Enable or disable recognition of `typename`. The default value is supplied by `DEFAULT_TYPENAME_ENABLED`. This option is valid only in C++ mode.

`--type_traits_helpers`
`--no_type_traits_helpers` Enable or disable support for type traits helpers (like `__is_union` and `__has_virtual_destructor`; intended to ease the implementation of ISO/IEC TR 19768). Only valid in C++. Type traits helpers are enabled in all C++ modes by default, except in GNU and Sun C++ modes.

`--uliterals`
`--no_uliterals` Enable or disable recognition of U-literals (string literals of the forms `U"..."` and `u"..."`, and character literals of the forms `U'...'` and `u'...'`). In C++ mode, this also enables or disables the C++11 `char16_t` and `char32_t` keywords.

`--variadic_macros`
`--no_variadic_macros` Enable or disable support for variadic macros (a C99 feature that is also available in C++ mode).

`--extended_variadic_macros`
`--no_extended_variadic_macros` Enable or disable support for “extended variadic macros,” an extension that emulates the behavior of certain other C compilers when it comes to variadic macros.

`--variadic_templates`
`--no_variadic_templates` Enable or disable support for “variadic templates,” a C++11 feature that allows declaration of templates with a variable number of arguments. The default value is dependent on the major language mode and on `DEFAULT_VARIADIC_TEMPLATES_ENABLED`. This option is valid only in C++ modes.

`--vla`
`--no_vla` Enable or disable support for “variable length arrays,” a C99 feature and an extension in some other modes that allows the declaration and use of arrays of automatic storage duration with dimensions that are fixed at run time. These options are available only if `VLA_ALLOWED` is configured to `TRUE`.

`--wchar_t_keyword`
`--no_wchar_t_keyword` Enable or disable recognition of `wchar_t` as a keyword. The default value is supplied by `DEFAULT_WCHAR_T_IS_KEYWORD`. This option is valid only in C++ mode. The front end can be configured to define a preprocessing variable when `wchar_t` is recognized as a keyword. This preprocessing variable may then be used by the standard header files to determine whether a typedef should be supplied to define `wchar_t`.

1.1.6 Alternative Language Behaviors

The following options affect the meaning of various language constructs.

`--arg_dep_lookup`
`--no_arg_dep_lookup`

1.1 Command Line

In C++, controls whether argument dependent lookup of unqualified function names is performed

`--base_assign_op_is_default`

`--no_base_assign_op_is_default`

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. This option is enabled by default in cfront compatibility mode.

`--class_name_injection`

`--no_class_name_injection`

In C++, 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).

`--const_string_literals`

`--no_const_string_literals`

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).

`--default_calling_convention` *calling-convention*

The calling convention that should be assumed for functions that are declared without an explicit calling convention. This option may only be used if `MICROSOFT_EXTENSIONS_ALLOWED` is `TRUE`. *calling-convention* must be one of `__cdecl`, `__fastcall`, `__stdcall`, and `__thiscall`.

`--default_nocommon_tentative_definitions`

`--default_common_tentative_definitions`

Control whether tentative definitions are placed in “common” storage. These options are only available in configurations with `GNU_EXTENSIONS_ALLOWED` set to `TRUE`. The default can be overridden for specific variables with the GNU attributes “common” and “nocommon”.

`--defer_parse_function_templates`

`--no_defer_parse_function_templates`

Enable or disable deferral of prototype instantiations until the first actual instantiation of a function. This is used to permit the compilation of programs that contain definitions of unusable function templates. It is enabled by default in g++ mode when `gnu_version` is `>= 30400` (but not when using the C++-generating back end).

`--dep_name`

`--no_dep_name`

Enable or disable dependent name processing; i.e., the special lookup of names used in templates as required by the C++ standard. The default value is supplied by `DEFAULT_DEPENDENT_NAME_PROCESSING`. This option is valid only in C++ mode and `--dep_name` cannot be combined with `--no_parse_templates`.

`--distinct_template_signatures`

`--no_distinct_template_signatures`

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. (This option is available only on versions that do name mangling on the names of external entities.) 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.

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

`--far_data_pointers`
`--near_data_pointers`
`--far_code_pointers`
`--near_code_pointers` Set the default size for pointers when support for near and far is enabled (e.g., in Microsoft 16-bit mode). Ignored in other modes.

`--friend_injection`
`--no_friend_injection` In C++, 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.

`--gcc89_inlining` Use the GNU C89 semantics of `inline` in C99 mode. (Not valid in C++ modes.)

`--guiding_decls`
`--no_guiding_decls` 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. Default behavior is specified by `DEFAULT_GUIDING_DECLS_ALLOWED`. This option is valid only in C++ mode.

`--ignore_std` Enable a g++ compatibility feature that makes the `std` namespace a synonym for the global namespace.

`--implicit_extern_c_type_conversion`
`--no_implicit_extern_c_type_conversion`

1.1 Command Line

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: `IMPL_CONV_BETWEEN_C_AND_CPP_FUNCTION_PTRS_POSSIBLE` must be set for it to be available. (It is useful for cfront compatibility when `DEFAULT_C_AND_CPP_FUNCTION_TYPES_ARE_DISTINCT` is `TRUE` — in standard C++ the linkage specification is part of the function type, with the consequence that otherwise identical function types, one declared extern "C" and the other declared extern "C++", are viewed as distinct.)

`--implicit_typename`

`--no_implicit_typename`

Enable or disable implicit determination, from context, whether a template parameter dependent name is a type or nontype. The default value is supplied by `DEFAULT_IMPLICIT_TYPENAME_ENABLED`. This option is valid only in C++ mode.

`--late_tiebreaker`

`--early_tiebreaker`

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” tiebreaker 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.

`--long_lifetime_temps`

`--short_lifetime_temps`

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).

`--long_preserving_rules`

`--no_long_preserving_rules`

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, not the rules used by the pcc compiler. The significant difference is in the handling of “long *op* unsigned int” when int and long are the same size. The ANSI/ISO/pcc 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).

`--nonconst_ref_anachronism`

`--no_nonconst_ref_anachronism`

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.

`--nonstd_default_arg_deduction`

`--no_nonstd_default_arg_deduction`

Controls whether default arguments are retained as part of deduced function types. The C++ standard requires that default arguments not be part of deduced function types.

`--nonstd_instantiation_lookup`

`--no_nonstd_instantiation_lookup`

Controls whether the lookup of names during template instantiation should, instead of the normal lookup rules, use rules that were part of the C++98 working paper for some time during the development of the standard. In this mode, names are looked up in both the namespace of the template definition and in the namespace in which a template entity was first referenced in a way that would require an instantiation.

`--nonstd_qualifier_deduction`

`--no_nonstd_qualifier_deduction`

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.

`--old_for_init`

`--new_for_init`

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. The default behavior is specified by configuration constant `DEFAULT_USE_NONSTANDARD_FOR_INIT_SCOPE`. This option is valid only in C++ mode.

`--pack_alignment n`

Set the default alignment for packing classes and structs to `n`, a power-of-2 value within the range defined by `TARG_MINIMUM_PACK_ALIGNMENT` and `TARG_MAXIMUM_PACK_ALIGNMENT`. The argument `n` is the default maximum alignment for nonstatic data members; it can be overridden by a `#pragma pack` directive. This option is allowed only when `USER_CONTROL_OF_STRUCT_PACKING` is `TRUE`.

`--parse_templates`

`--no_parse_templates`

Enable or disable the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). It is done by default if dependent name processing is enabled. This option is valid only in C++ mode.

`--short_enums`

Force all enumeration types to be “packed” (meaning that the underlying type of the enumeration is chosen to be the smallest integer that will accommodate the enumerator constants). Only valid in GNU C compatibility mode.

`--signed_bit_fields`

Make bit fields declared with a plain integer type (e.g., `short` or `int`, but not signed `short` or unsigned `int`) signed.

`--signed_chars`

`-s`

Make plain `char` signed. The default signedness for `char` is selected at the time of installation of the front end (information on this appears in a later section of

1.1 Command Line

this document). When plain char is signed, the macro `__SIGNED_CHARS__` is defined by the front end.

`--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. 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
}
```

As of July 1996, the above is again acceptable, if `ptrdiff_t` is configured as long. Using a parameter of type `ptrdiff_t` (instead of `unsigned int`) is recommended for portability.

`--no_stdarg_builtin`

Disable special treatment of the `<stdarg.h>` header. When enabled, the `<stdarg.h>` header is treated as a built-in, and references to its macros (`va_start` et al) are passed through as such in generated C or C++ code. The default value is supplied by `DEFAULT_PASS_STDARG_REFERENCES_TO_GENERATED_CODE`. This option cannot be used when the feature is disabled by default.

`--unsigned_bit_fields`

Make bit fields declared with a plain integer type (e.g., `short` or `int`, but not signed `short` or `unsigned int`) unsigned.

`--unsigned_chars`

`-u`

Make plain `char` unsigned. The default signedness for `char` is selected at the time of installation of the front end (information on this appears in a later section of this document).

`--using_std`

`--no_using_std`

Enable or disable implicit use of the `std` namespace when standard header files are included. The default behavior is specified by the configuration constant `DEFAULT_IMPLICIT_USING_STD`. This option is valid only in C++ mode. Note that this does *not* do the equivalent of putting a “`using namespace std;`” in the program to allow old programs to be compiled with new header files; it has a special and localized meaning related to the EDG versions of certain header files, and is unlikely to be of much use to end-users of the Front End.

1.1.7 Template Instantiation Options

`--auto_instantiation`
`--no_auto_instantiation`
`-T` Enable or disable automatic instantiation of templates. The `-T` option is equivalent to `--auto_instantiation`. The default behavior is specified by the configuration flag `DEFAULT_AUTOMATIC_INSTANTIATION_MODE`. See the section of this chapter on template instantiation. This option is valid only in C++ mode.

`--definition_list_file file_name`
 This option specifies the name of the template definition list file passed between the front end and the prelinker. This file is a temporary file and does not remain after the compilation is complete. This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.

`--exported_template_file file-name`
 This option specifies the name to be used for the exported template file used for processing of exported templates. This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.

`--ii_file file-name`
 This option specifies the name to be used for the template instantiation request file used in automatic instantiation mode. This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.

`--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. `-B` is equivalent to `--implicit_include`. The default behavior is specified by the configuration flag `DEFAULT_IMPLICIT_TEMPLATE_INCLUSION_MODE`. See the section of this chapter on template instantiation. This option is valid only in C++ mode.

`--instantiate mode`
`-tmode` Control instantiation of external template entities. External template entities are external (i.e., 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):

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

See the section of this chapter on template instantiation. This option is valid only in C++ mode.

`--instantiation_dir dir-name`

1.1 Command Line

When `--one_instantiation_per_object` is used, this option can be used to specify a directory into which the generated object files (or C files, if the C-generating back end is being used) should be put. The `eccp` script always specifies this option when invoking the front end, with the default of `Template.dir` if the option is not specified by the user.

`--one_instantiation_per_object`

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, i.e., everything that isn't 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. This option is valid only in C++ mode, and only if the configuration constant `ONE_INSTANTIATION_PER_OBJECT` is `TRUE`.

`--pending_instantiations=n`

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. The default is specified by the configuration parameter `DEFAULT_MAX_PENDING_INSTANTIATIONS`.

`--suppress_instantiation_flags`

When automatic instantiation is being used, this option suppresses the generation of the special symbols used by the prelinker to do automatic instantiation processing. This option is for use by the driver and prelinker and should not be used directly by users. This option is used by to implement the driver option that removes the instantiation flags from object files once the prelinking step has completed.

`--template_directory dir`

Specifies a directory name to be placed on the exported template search path. The directories are used to find the definitions of exported templates and are searched in the order in which they are specified on the command-line. The current directory is always the first entry on the search path.

`--template_info_file file-name`

This option specifies the name to be used for the template instantiation information file used in automatic instantiation mode. This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.

1.1.8 Code Generation Options

The following options control the IL produced, the C-generating back end, or the C++-generating back end.

`--no_code_gen`

`-n`

Do syntax-checking only, i.e., do not run the back end.

`--force_vtbl`

Force definition of virtual function tables in cases where the heuristic used by the front end to decide on definition of virtual function tables provides no guidance. See `--suppress_vtbl`. This option is valid only in C++ mode.

<code>--no_il_lowering</code> <code>-N</code>	On versions that include IL lowering and that write an intermediate language file, suppress both IL lowering and the back end, and write an unlowered (i.e., C++ rather than C) IL file. Probably not of interest to end users.
<code>--inlining</code> <code>--no_inlining</code>	Enable or disable minimal inlining of function calls. These options are available only when <code>MINIMAL_INLINING</code> is configured to be <code>TRUE</code> . The default is to do inlining when the C-generating back end is used.
<code>--module_init name1,name2,...</code> <code>-iname1,name2,...</code>	When generating C output for testing (see <code>c_gen_be</code> in an Appendix), when compiling in C mode, and if the current compilation contains a main program, force calling of the file-scope initialization routines in the indicated compiled units. This is needed only when there are file-scope union initializations in translation units that do not contain a main program. A message will be issued during compilation of such units, indicating the need for the <code>--module_init</code> option. All necessary names must be specified in one <code>--module_init</code> option; a second <code>--module_init</code> will overwrite the original list, not add to it. This option is not needed when object code is generated by a back end other than the C-generating back end, nor is it needed when the C-generating back end is configured to generate ANSI C. This option is not needed in C++ mode, because C++ has a language mechanism for getting initialization code executed at program start-up.
<code>--msvc_target_version version-number</code>	When using the C-generating back end to produce code that is to be compiled with the Microsoft compiler, this specifies the version of the Microsoft compiler being used (e.g., 1300 is used for MSVC version 7). This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.
<code>--remove_unneeded_entities</code> <code>--no_remove_unneeded_entities</code>	Enable or disable an optimization to prune the IL tree of types, variables, routines, and related IL entries that are not “really needed.” (Something may be referenced but unneeded if 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.) The positive form of the option is available only when <code>MAINTAIN_NEEDED_FLAGS</code> is <code>TRUE</code> .
<code>--suppress_vtbl</code> <code>-V</code>	Suppress definition of virtual function tables in cases where the heuristic used by the front end 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 option <code>--suppress_vtbl</code> suppresses the definition of the virtual function tables for such classes, and <code>--force_vtbl</code> forces the definition of the virtual function table for such classes. <code>--force_vtbl</code> differs from the default behav-

1.1 Command Line

ior in that it does not force the definition to be local. This option is valid only in C++ mode.

1.1.9 Precompiled Header Options

The following options control how precompiled header files are generated and used.

<code>--pch</code>	Automatically use and/or create a precompiled header file — for details, see the “Precompiled Headers” section in this chapter. If <code>--use_pch</code> or <code>--create_pch</code> (manual PCH mode) appears on the command line following this option, its effect is erased.
<code>--create_pch file-name</code>	If other conditions are satisfied (see the “Precompiled Headers” section), create a precompiled header file with the specified name. If <code>--pch</code> (automatic PCH mode) or <code>--use_pch</code> appears on the command line following this option, its effect is erased.
<code>--use_pch file-name</code>	Use a precompiled header file of the specified name as part of the current compilation. If <code>--pch</code> (automatic PCH mode) or <code>--create_pch</code> appears on the command line following this option, its effect is erased.
<code>--pch_dir directory-name</code>	The directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (<code>--pch</code>) or with manual PCH mode (<code>--create_pch</code> or <code>--use_pch</code>).
<code>--pch_messages</code> <code>--no_pch_messages</code>	Enable or disable the display of a message indicating that a precompiled header file was created or used in the current compilation.
<code>--pch_verbose</code>	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.
<code>--pch_mem size</code>	The number of 1024-byte units of preallocated memory in which to save precompiled-header state (only if <code>USE_MMAP_FOR_MEMORY_REGIONS</code> is <code>FALSE</code>).

1.1.10 C++/CLI Options

<code>--mscorlib_file_name file-name</code>	Use the given assembly metadata file instead of <code>mscorlib.dll</code> to import system types.
<code>--preusing file-name</code>	Import the given assembly metadata file prior to processing the input source (but after importing <code>mscorlib.dll</code> or the file specified with the <code>--mscorlib_file_name</code> option).
<code>--using_directory directory-name</code>	Add the given directory to the search path used when searching for assembly metadata files.
<code>--using_framework_directory</code> <code>--no_using_framework_directory</code>	

Do or do not implicitly include the installation directory of the Microsoft .NET runtime in the search path used when searching for assembly metadata files. The default is to include that directory.

1.1.11 Front End Debugging Options

`--dbn`
`--dbname=n`
`-dn`
`-dname=n`

Set the debug output level in the front end, which controls the output of flow-tracing and similar information to the debug output file (`stderr`). `--dbn` sets the debug level to *n* (in the range 1–5) for the entire execution of the compiler (higher values generate more voluminous debugging output). One can control the debug output more precisely by specifying a routine name and the debug level desired upon entry to that routine, as in `--dbname=n`. Regardless of the previous setting of the debug level, it will be changed to the indicated value on entry to the indicated routine; on exit, it will be restored to its former value. One can also use `+=` or `-=` in place of `=` in that option to increase or decrease the debug level relative to its current value. The former value is also restored on exit from the routine in those cases. Ordinarily, a message is written to debug output whenever the debug level is changed (i.e., both on entry and exit in the above cases). If that is not desirable, a `!` (implying “not”) can be appended to the level number in the command-line option, to indicate that the level-change messages should not be produced for that request. Several routine names can be specified, and/or several requests can be made, by separating them with commas, as in the following:

```
--db fe_init,fe_wrapup+=2,get_token=3!
```

`-d-name1, -name2, ...`

Enable the named “debug flags”, which will enable debug output associated with the flag. For example, `-d-dump_layout` produces output related to the class type layout process.

`-d#name1, #name2, ...`

Disable the named “debug flags”.

`--db_name=name`

Enable debug output for source entities with the given name. Not all debug output takes this option into account; i.e., even with this option some debug output not associated with the given name may be produced. When simultaneously compiling multiple translation units, entities in all translation units are considered by default. The given name can be restricted to entities in the primary translation units by preceding it with `[]` or to entities in a secondary translation unit by preceding it with `[file]` where *file* is the name of the secondary source file. For example,

```
-d-trans_corresp --db_name=[x.cpp]A::f
```

will output debugging information about cross-translation-unit correspondences for a member `A::f` declared in a secondary translation unit defined by source file `x.cpp`.

1.2 Environment Variables

1.1.12 Miscellaneous Options

<code>--version</code>	
<code>-v</code>	Display the version number of the front end, and the Edison Design Group copyright.
<code>--building_runtime</code>	Used to indicate that the EDG runtime library is being compiled. This causes additional macros to be predefined that are used to pass target configuration information from the front end to the runtime.
<code>--edg_base_dir</code>	The name of the directory in which to find files needed by the front end at execution time. This option is supplied for use by the driver program that invokes the front end and is not intended to be used by end-users.
<code>--dump_configuration</code>	Display the complete set of configuration macros with which the front end was built.. The output (to the error output file) can be captured and used as the contents of the <code>defines.h</code> file to recreate that configuration. This option is available only in executables built with <code>DEBUG</code> set to <code>TRUE</code> .

1.2 Environment Variables

The environment variable `USR_INCLUDE` can be set to a directory to be used instead of `/usr/include` on the standard include file search list. (Of course, this has no effect if the front end has been configured to have an empty “standard list” of include files.)

When the front end is built with the `__MICROSOFT_OS__` flag set, the environment variable `TMP` is used to specify the directory to be used for temporary files. If `TMP` is not set, and when `__MICROSOFT_OS__` is not set, the environment variable `TMPDIR` is used to indicate a directory to be used for temporary files. If `TMPDIR` is not set, a default temporary directory (often, `/usr/tmp`) is used. (Note: at the present time, the front end only uses temporary files when generating an intermediate language file for immediate use by a back end called in the same program, and in the C-generating back end.)

1.3 Diagnostic Messages

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

- Catastrophic 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 or 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).

Diagnostics are written to the error output file with a form like the following:

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

Diagnostics are normally written to `stderr`, but the front end can be configured to use `stdout` by setting the `DIRECT_ERROR_OUTPUT_TO_STDOUT` configuration macro. When this flag is set, messages are directed to `stdout`, except when doing preprocessing only. Errors during command-line processing are always directed to `stderr`.

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 front end can be configured so that the severity string is omitted when the severity is “*error*”.

A command line option 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.

A command line option 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. If the severity may be overridden, the error number will include the suffix “-D” (for “discretionary”); otherwise no suffix will be present.

```
"Test_name.c", line 7: error #64-D: declaration does not declare anything
    struct {};
    ^
"Test_name.c", line 9: error #77: this declaration has no storage class or
    type specifier
    xxxxx;
    ^
```

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.c", 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 front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

1.4 Termination Messages

```
"test.c", 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.

If the front end is configured with `FULLY_RESOLVED_MACRO_POSITIONS` (and, optionally, `MACRO_INVOCATION_TREE_IN_IL`) set to `TRUE`, messages referring to text occurring inside a macro expansion may provide additional information. This additional information gives the source location from which the text was copied into the macro expansion (i.e., from a macro definition or one of the arguments in the top-level macro invocation), as well as possibly the chain of macro invocations leading to the expanded text to which the diagnostic refers. If the front end is configured to permit the display of this information, it can also be enabled or disabled by the command-line option `--[no_]macro_positions_in_diagnostics`. The information is presented in a form like the following:

```
"test.c", line 1: error: identifier "val" is undefined
  #define incr(x) (x)+=val
                    ^
    in expansion of macro "incr" at "test.c", line 2,
    in expansion of macro "next_val" at "test.c", line 4:
  return next_val(i);
        ^
```

1.4 Termination Messages

When the front end is installed without the `USING_DRIVER` option (see information on configuration later in this document), it writes sign-off messages to the error output file. 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, above); 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 and should be reported to those responsible for its maintenance.

1.5 Response to Signals

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

1.6 Exit Status

On completion, the front end returns with a code indicating the highest-severity diagnostic detected: 4 if there was a catastrophic error, 2 if there were any errors, or 0 if there were any warnings or remarks or if there were no diagnostics of any kind.

If multiple source files are compiled, the exit status indicates the highest-severity diagnostic detected in the entire compilation.

1.7 Finding Include Files

A file name specified in a `#include` directive is searched for in a set of directories specified by command-line options and environment variables. If the file name specified does not include a suffix, a set of suffixes is used when searching for the file.

Files whose names are not absolute pathnames and that are enclosed in `" . . . "` will be searched for in the following directories, in the order listed:

1. The directory containing the current input file (the primary source file or the file containing the `#include`);¹
2. any directories specified in `--include_directory` options (in the order in which they were listed on the command line);
3. any directories on the standard list (this list is selected at the time of installation of the front end; often, it is just the one directory `/usr/include` or a similar system include directory for C++).

For file names enclosed in `< . . . >`, only the directories that are specified using the `--include_directory` option and those on the standard list are searched. If the directory name is specified as `"-"`, e.g., `"-I-"`, the option indicates the point in the list of `--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 `--include_directory` options following the `-I-`, and the directories of item 3 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 a default include directory has been specified using the `USR_INCLUDE` configuration flag or environment variable, it is considered a system include directory.

If the file name 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 specified by the `DEFAULT_INCLUDE_FILE_SUFFIX_LIST` configuration parameter. 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 (i.e., the front end will always attempt to add a suffix from the suffix list when the file name has no suffix).

¹ However, if `STACK_REFERENCED_INCLUDE_DIRECTORIES` is `TRUE`, the directories of all the source input files currently in use are searched, in reverse order of `#include` nesting.

1.8 C++ Dialect Accepted

In strict C++ mode, the front end accepts the complete C++ language as defined by the ISO/IEC 14882:2003 standard, including export templates (if the front end's configuration allows them). In nonstrict C++ mode (which is the default in common configurations), a small set of features—particularly, export templates—are not supported, but some minor extensions commonly available on other compiler are accepted.

Many features of the newer standard, ISO/IEC 14882:2011, also known as C++11 or C++0x, are also implemented (enabled through the command-line option `--c++11`). Eventually, the language described by that standard will also be completely accepted.

Extensive compatibility modes are also available for the Microsoft Visual C++ compilers and the GNU C++ (“GCC”) compilers. A more basic Sun C++ compatibility mode is also implemented.

The front end 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 EDG front end may produce a different error or no error at all.

Command-line options are available to enable and disable anachronisms, to control strict standard-conformance checking, and to adjust the severity of certain diagnostics.

1.8.1 C++11 Language Features Accepted

The following features added in the C++11 standard are enabled in C++11 mode. This mode can be combined with the option for strict standard conformance. Several of these features are also enabled in default (nonstrict) C++ mode.

- A “right shift token” (`>>`) can be treated as two closing angle brackets. For example:

```
template<typename T> struct S {};  
S<S<int>> s; // Okay.  
           // No whitespace needed between closing angle brackets.
```

- The `static_assert` construct is supported. For example:

```
template<typename T> struct S {  
    static_assert(sizeof(T) > 1, "Type T too small");  
};  
S<char[2]> s1; // Okay.  
S<char> s2;   // Instantiation error due to failing static_assert.
```

- The friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:

```
typedef struct S ST;  
class C {  
    friend S;           // Okay (requires S to be in scope).  
    friend ST;         // Okay (same as "friend S;").  
    friend int;        // Okay (no effect).  
    friend S const;    // Error: cv-qualifiers cannot appear directly.  
};
```

- Local and unnamed types (and types based on such types) can be used for template type arguments. They can also be used in the signatures of functions, if they appear there through template substitution.

- Mixed string literal concatenations are accepted (a feature carried over from C99):


```
wchar_t *str = "a" L"b"; // Okay, same as L"ab".
```
- C99 preprocessor extensions are carried over. Variadic macros and empty macro arguments are accepted. The `__STDC_HOSTED__` macro is predefined (to a configuration-dependent value).
- The C99-style `_Pragma` operator is supported.
- In function bodies, the reserved identifier `__func__` refers to a predefined array containing a string representing the function's name (a feature carried over from C99).
- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):


```
enum E { e, };
```
- If `LONG_LONG_ALLOWED` is `TRUE`, the type `long long` is accepted. Unsuffixed integer literals that cannot be represented by type `long`, but could potentially be represented by type `unsigned long`, have type `long long` instead (this matches C99, but not the treatment of the `long long` extension in C89 or default C++ mode).
- An explicit instantiation directive may be prefixed with the `extern` keyword to suppress the instantiation of the specified entity.
- `export` templates are disabled by default (even in strict C++11 mode with configurations that otherwise enable that C++03 feature).
- The keyword `typename` followed by a qualified-id can appear outside a template declaration.


```
struct S { struct N {}; };
typename S::N *p; // Silently accepted in C++11 mode.
```
- The keyword `auto` can be used as a type specifier in the declaration of a variable or reference. In such cases, the actual type is deduced from the associated initializer. This feature can be used for variable declarations, for in-class declarations of static const members, and for new-expressions.


```
auto x = 3.0; // Same as "double x = 3.0;"
auto p = new auto(x); // Same as "double *p = new double(x);"
struct S {
    static auto const m = 3; // Same as "static int const m = 3;"
};
```

By default, `auto` is no longer accepted as a storage class specifier (but an option is available to re-enable it).
- Trailing return types are allowed in top-level function declarators. These must be paired with the `auto` type specifier.


```
auto f()->int*; // Same as: int *f();
```
- The keyword `decltype` is supported: It allows types to be described in terms of expressions. For example:


```
template<typename T> struct S {
    decltype(f(T())) *p; // A pointer to the return type of f.
};
```
- The constraints on the code points implied by universal character names (UCNs) are slightly different: UCNs for surrogate code points (0xD000 through 0xDFFF) are never permitted, and UCN corresponding to control characters or to characters in the basic source character set are permitted in string literals.
- Scoped enumeration types (defined with the keyword sequence `enum class`) and explicit underlying integer types for enumeration types are supported. For example:

1.8 C++ Dialect Accepted

```
enum class Primary { red, green, blue };
enum class Danger { green, yellow, red }; // No conflict on "red".
enum Code: unsigned char { yes, no, maybe };
void f() {
    Primary p = Primary::red; // Enum-qualifier is required to access
                             // scoped enumerator constants.
    Code c = Code::maybe; // Enum qualifier is allowed (but not required)
} // for unscoped enumeration types.
```

- Lambdas are supported. For example:

```
template<class F> int z(F f) { return f(0); }
int g() {
    int v = 7;
    return z([v](int x)->int { return x+v; });
}
```

- Rvalue references are supported. For example:

```
int f(int);
int &&rr = f(3);
```

- Functions can be “deleted”. For example:

```
int f(int) = delete;
short f(short);
int x = f(3); // Error: selected function is deleted.
int y = f((short)3); // Okay.
```

- Special member functions can be explicitly “defaulted” (i.e., given a default definition). For example:

```
struct S { S(S const&) = default; };
struct T { T(T const&); };
T::T(T const&) = default;
```

- Conversion functions can be marked `explicit` to indicate that they should only be considered for explicit conversions, and in certain contexts that require a boolean value (like the controlling expression for an if-statement).
- The operand of `sizeof`, `typeid`, or `decltype` can refer directly to a non-static data member of a class without using a member access expression. For example:

```
struct S {
    int i;
};
decltype(S::i) j = sizeof(S::i);
```

- The keyword `nullptr` can be used as both a null pointer constant and a null pointer-to-member constant. Variables and other expressions whose type is that of the `nullptr` keyword (conventionally known by its standard typedef, `std::nullptr_t`) can also be used as null pointer(-to-member) constants, although they are only constant expressions if they otherwise would be. For example:

```
#include <cstddef> // To get std::nullptr_t
struct S { };
template <int *> struct X { };
std::nullptr_t null();
void f() {
    void *p = nullptr; // Initializes p to null pointer
    int S::* mp = nullptr; // Initializes mp to null ptr-to-member
```

```

    p = null();           // Sets p to null pointer
    X<nullptr> xnull0;    // Instantiates X with null int * value
    X<null()> xnull1;    // Error: template argument not a
                        // constant expression
}

```

- Attributes delimited by double square brackets ([[...]]) are accepted in declarations. The standard attributes `noreturn` and `carries_dependency` are supported. For example:

```
[[noreturn]] void f();
```

- Alias and alias template declarations are supported. For example:

```

using X = int;
X x; // equivalent to "int x"
template <typename T> using Y = T*;
Y<int> yi; // equivalent to "int* yi"

```

- Variadic templates are supported. For example:

```

template<class ...T> void f(T ...args) {
    int i = sizeof...(args);
}
int main() {
    f(1, 2, 3, 4);
}

```

- U-literals as well as the `char16_t` and `char32_t` keywords are supported. For example:

```

char16_t *str = u"A 16-bit character string";
char32_t ch = U'\U00012345'; // A 32-bit character literal

```

- Many errors in expressions that arise during the substitution of template parameters in function templates are now treated as deduction failures rather than definite errors. This may result in a valid program if another (overloaded) function template allows the substitution. This is also known as “SFINAE for expressions” (where SFINAE stands for “Substitution Failure Is Not An Error”): In the original C++ standard (1998, 2003) SFINAE was mostly limited to simple type substitutions.
- Access checking of names used as base classes is done in the context of the class being defined. For example:

```

class B { protected: class N {} };
class D: B::N, B {}; // now allowed

```

1.8.2 Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled:

- `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.

1.8 C++ Dialect Accepted

- 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 nonnested 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; }
```

It will be noted 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 nonconst 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
}
```

1.8.3 “Default” C++ Mode

The following extensions are accepted default C++ mode. Most of these are also accepted in any other C++ mode that does not diagnose strict ANSI violations as errors (exceptions are explicitly noted).

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

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

- Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

```
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();
};
```

- If `ALLOW_NONSTANDARD_ANONYMOUS_UNIONS` is `TRUE`, an extension is supported to allow an anonymous union to be introduced into a containing class by a `typedef` name — it needn't be declared directly, as with a true anonymous union. For example:

```
typedef union {
    int i, j;
} U; // U identifies a reusable anonymous union.
class A {
    U; // Okay -- references to A::i and A::j are allowed.
};
```

In addition, the extension also permits “anonymous classes” and “anonymous structs,” as long as they have no C++ features (e.g., no static data members or member functions and no nonpublic members) and have no nested types other than other anonymous classes, structs, or unions. For instance,

```
struct A {
    struct {
        int i, j;
    }; // Okay -- references to A::i and A::j are allowed.
};
```

- If recognition of the `restrict` keyword is enabled, the C99 `restrict` feature is supported in a form extended for C++, which allows `restrict` as a type qualifier for reference and pointer-to-member types and for nonstatic member functions. The set of C++ extensions is described in J16/92-0057.
- If `IMPL_CONV_BETWEEN_C_AND_CPP_FUNCTION_PTRS_POSSIBLE` is `TRUE`, 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 (though it is pointless unless `DEFAULT_C_AND_CPP_FUNCTION_TYPES_ARE_DISTINCT` is `TRUE`). When `DEFAULT_IMPL_CONV_BETWEEN_C_AND_CPP_FUNCTION_PTRS_ALLOWED` is set, it is enabled by default; it can also be enabled in cfront-compatibility mode or with command-line option `--implicit_extern_c_type_conversion`.

- 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";
```

- 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).
- Nonstatic local variables of an enclosing function can be referenced in a non-evaluated expression (e.g., a `sizeof` expression) inside a local class. A warning is issued.

1.8 C++ Dialect Accepted

- In default C++ mode (but not other non-C++11 modes), the friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:

```
typedef struct S ST;
class C {
    friend S;           // Okay (requires S to be in scope).
    friend ST;         // Okay (same as "friend S;").
    friend int;        // Okay (no effect).
    friend S const;    // Error: cv-qualifiers cannot appear directly.
};
```

- In default C++ mode, mixed string literal concatenations are accepted. (This is a feature carried over from C99 and also available in GNU modes. It is not enabled in other non-C++11 modes.)

```
wchar_t *str = "a" L"b"; // Okay, same as L"ab".
```

- In default C++ mode, variadic macros are accepted. (This is a feature carried over from C99 and also available in GNU modes. It is not by default enabled in other non-C++11 modes.)
- In default C++ mode, empty macro arguments are accepted (a feature carried over from C99).
- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):

```
enum E { e, };
```

Except where noted, all of the extensions described in the C dialect section are also allowed in C++ mode.

1.8.4 GNU C++ Mode

The front end can be built to accept GNU extensions. In GNU C++ mode, a large number of extensions and bugs are closely emulated. This feature enables the front end to compile many large and complex projects that were developed using the GNU tool chain; this includes major open-source projects.

Support for this emulation is ongoing: New GNU C++ features are added as needed, with priority given to features used in system headers.

Because the GNU compiler frequently changes behavior between releases, the front end provides an option (`--gnu-version`) to specify a specific version of GCC to emulate. Generally speaking, features and bugs are emulated to exactly match each known version of GCC, but occasionally the emulation is approximate and in such cases the front end is often a little more permissive than GCC on the principle that it is more important to accept source that GCC accepts than to diagnose every case that GCC diagnoses. The front end does not, however, attempt to emulate every GCC command-line option; in particular, GCC options to be extra-permissive are not emulated (however, the severity of specific error diagnostics can sometimes be decreased to accept constructs that are not by-default allowed in GNU emulation mode).

Some GNU features are only emulated when specific configuration options are enabled. For example, support for complex floating-point types is only available when the front end is configured to support C99 extensions.

The most recent versions of GCC implement some C++11 features that the front end does not yet implement. Most notably among these are “initializer list” extensions and `constexpr` functions. Support for these and other C++11 features will be added in forthcoming releases.

A few GCC extensions there are likely *not* going to be supported in the foreseeable future:

- The forward declaration of function parameters (so they can participate in variable-length array parameters).

- GNU-style complex *integral* types (complex *floating-point* types are supported)
- Nested functions
- Local structs with variable-length array fields. Such fields are treated (with a warning) as zero-length arrays in GNU C mode, which is a useful approximation in some circumstances, but not generally equivalent to the GNU feature.

1.8.5 Microsoft C++ Mode

The front end also extensively emulates the Microsoft Visual C++ compilers. As is the case with GNU modes, an option (`--microsoft_version`) is available to select a specific version of Microsoft's compiler to emulate.

Although Microsoft C++ and C modes generally emulate both extensions/features and bugs, the emulation of a certain class of bugs (considered more severe) can be controlled separately (using the `--no_microsoft_bugs/--microsoft_bugs` options).

Microsoft C++ mode can be combined with certain other language options to enable a larger superset of the language accepted by the Microsoft compiler. For example, C++11 mode and Microsoft C++ mode can be combined.

Support for Microsoft C++ emulation is ongoing: New features are added as needed, with priority given to features used in system headers.

1.8.6 Microsoft C++/CLI Mode

The front end implements the language extensions known as “C++/CLI” that simplifies writing C++-like programs for Microsoft's “.NET” environment. These extensions are formally described through the ECMA-372 standard, but where the Microsoft compiler deviates from the standard, the front end follows the Microsoft compiler rather than the standard.

Almost all C++/CLI features are implemented in the front end. This includes various managed class kinds and the special members (e.g. properties) that they can contain, delegates, handles and tracking references, `gcnew`, generics, importing of assemblies, hide-by-sig lookup rules, string literal rules, and so forth. A notable exception is Microsoft-style attributes (delimited by square brackets): The front end can parse them, but does not currently implement their semantics.

An earlier attempt at integrating C++ and .NET was known as “Managed C++”: The front end does not support those earlier Microsoft compiler features (which have since been deprecated by Microsoft).

1.8.7 Extensions Accepted in Sun Compatibility Mode

The front end provides a “Sun Compatibility” mode. This mode automatically selects a set of command-line options that provides the greatest compatibility with the Sun CC 5.x compiler. It also enables certain extensions that are accepted by the Sun compiler. Note that this mode is provided to enable the front end to compile code such as commonly used Sun header files, and is not intended to provide complete compatibility with the Sun compiler.

Sun compatibility mode currently corresponds to the following combination of options:

```
--no_guiding_decls
--extern_inline
--nonstd_using_decl
```

These settings can explicitly be overridden using more command-line options.

1.8 C++ Dialect Accepted

The following extensions are accepted in Sun compatibility mode:

- The Sun-specific link scope specifiers `__global`, `__symbolic`, and `__hidden` can be recognized. In modes that recognize these keywords, the pragmas “`#pragma disable_ldscope`” and “`#pragma enable_ldscope`” can be used to temporarily allow these keywords as normal identifiers. For example:

```
__global int i; // Accepted in some Sun modes.
#pragma disable_ldscope
int __global; // Okay: Keyword (temporarily) deactivated.
#pragma enable_ldscope
struct __global; // Error: __global is a keyword.
```

- The `__thread` specifier can be used to indicate that a variable should be placed in thread-local storage.
- In Sun mode, a non-tag from one scope hides a tag from another scope when both are visible as a result of a using-directive.

```
template <class T> struct B { };
namespace N {
    typedef B<char> A;
}
using namespace N;
class A;
typedef A C; // Sun mode uses typedef N::A instead of ambiguity
```

- A reference to an undeclared function template is accepted when the reference appears in the definition of a class template.

```
template <class T> struct A {
    friend void f<>(T&); // Accepted in Sun mode
};
```

- When a friend class is declared with an unqualified name, the lookup of that name is not restricted to the nearest enclosing namespace scope.

```
struct S;
namespace N {
    class C {
        friend struct S; // ::S in Sun and Microsoft modes
    }; // (N::S otherwise)
}
```

- The `typename` keyword is ignored in most contexts.

```
// The Sun compiler does recognize typename in a template parameter list
template <typename T> class C {
    // Illegal uses of typename that are accepted by the Sun compiler
    typename T t;
    typename typename T typename typename x;
};
```

- A default argument may be specified in the definition of a member function of a class template declared outside of the class.

```
template <class T> struct A {
    void f(int);
};
template <class T> void A<T>::f(int = 0) { }
```

- A template argument list may appear following a constructor name in constructor definition that appears outside of the class definition:

```
template <class T> struct A {
    A();
};
template <class T> A<T>::A<T>(){}
```

- Template parameters are (incorrectly) visible in class template specializations and specializations of members of class templates.

```
template <class T> struct A {
    void f();
};
template <> struct A<int> {
    T t; // T should not be visible here
};
template <> void A<char>::f() {
    T t; // T should not be visible here
}
```

- The explicit instantiation of a class does not cause the instantiation of its inline members.

```
template <class T> struct A {
    void f() { g(); } // Not instantiated in Sun mode
};
template class A<int>;
```

- An extern inline function that is referenced but not defined is permitted (with a warning).
- Default arguments are retained as part of deduced function types.
- When searching for a header file specified using the `<...>` syntax, the front end searches for the specified header file name and also for a version with the suffix `.SUNWCCh` appended (e.g., `stddef.h.SUNWCCh`).
- The `std` namespace is predeclared.
- A reference to a nonstatic data member may be written without an object (not even an implicit `this->`) inside a `sizeof`:

```
typedef struct _X {
    char A[512];
} X;
static unsigned int foo() {
    return sizeof(X::A);
}
```

- A storage class may appear in a declaration that also has a “direct” linkage specification — e.g.,

```
extern "C" static void f();
```

is treated as equivalent to

```
extern "C" { static void f(); }
```

1.8.8 Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled:

1.8 C++ Dialect Accepted

- `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 nonnested 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; }
```

It will be noted 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 nonconst 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
}
```

1.8.9 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. (This is an outright bug, which unfortunately happens to be exploited in the NIH class libraries.)
- When a builtin operator is considered alongside overloaded operators in overload resolution, the match of an operand of a builtin type against the builtin type required by the builtin 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 and is only enabled when the configuration switch `CFRONT_2_1_OBJECT_CODE_COMPATIBILITY` is `TRUE`.
- 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 but can be disabled by setting the parameter `CFRONT_GLOBAL_VS_MEMBER_NAME_LOOKUP_BUG` to `FALSE`. 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 nonstatic data member. Member type names don’t 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:

1.8 C++ Dialect Accepted

```
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
};

```

- The preprocessing symbol `cplusplus` is defined in addition to the standard `__cplusplus`.

1.8.10 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 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.

1.8 C++ Dialect Accepted

- 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(); // No warning with --microsoft_bugs
};
```

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

```
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 nonstatic 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
}
```

Note that 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 `xyz`, 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.
- 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.

1.8 C++ Dialect Accepted

```
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 the user has 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);
```

When `DEFAULT_C_AND_CPP_FUNCTION_TYPES_ARE_DISTINCT` is `FALSE`, `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.)

However, even if `DEFAULT_C_AND_CPP_FUNCTION_TYPES_ARE_DISTINCT` is `TRUE`, it is possible to set `IMPL_CONV_BETWEEN_C_AND_CPP_FUNCTION_PTRS_POSSIBLE` to `TRUE` as well, so that in cfront-compatibility mode an implicit type conversion will always be done between a pointer to an extern "C" function and a pointer to an extern "C++" function.

- 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
};
```

- Default arguments are retained as part of deduced function types.

1.9 C Dialect Accepted

The front end accepts the ANSI/ISO C language as defined by ANSI/ISO 9899:1990. In C99 mode, it accepts the ANSI/ISO C language defined by ISO/IEC 9899:1999 as modified by Technical Corrigenda 1 through 3. When Embedded C extensions are enabled, the front end can accept the constructs described in ISO/IEC TR 18037 (fixed-point types, named address space qualifiers, and named-register storage class specifiers).

The front end extensively emulates the C modes of the Microsoft and GNU compilers. See “GNU C++ Mode” on page 35 and “Microsoft C++ Mode” on page 36 for remarks that also apply to the corresponding C modes. These modes are continuously updated to emulate new features and bugs of the corresponding compilers.

There is no Sun C mode corresponding to Sun C++ mode, however.

The special comments recognized by the UNIX `lint` program — `/*ARGSUSED*/`, `/*VARARGS*/` (with or without a count of non-varying arguments), and `/*NOTREACHED*/` — are also recognized by the front end.

When Unified Parallel C (UPC) mode is enabled, it also accepts the core language extensions described in version 1.0 of the UPC Language Specifications (available from <http://upc.gwu.edu>).

1.9.1 C99 Features Available in Other Modes

Certain C language features were added in the C99 version of the ISO C standard and are supported in C99 mode. For a full list of those features, see ISO/IEC 9899:1999. A few features are noteworthy, however, because they are implementation-specific or because they can be enabled in older C modes or (when indicated) in C++ mode:

- The options `--variadic_macros`, `--no_variadic_macros`, `--extended_variadic_macros` and `--no_extended_variadic_macros` control whether macros taking a variable number of arguments are recognized. Configuration flags are available to select the default setting of these options; these are also available in C++ mode.

1.9 C Dialect Accepted

Ordinary variadic macros (as included in C99) are illustrated by the following example:

```
#define OVM(x, ...) x(__VA_ARGS__)
void f() { OVM(sprintf, "%s %d\n", "Three args for ", 1); }
/* Expands to: sprintf("%s %d\n", "Three args for ", 1) */
```

During expansion the special identifier `__VA_ARGS__` will be replaced by the trailing arguments of the macro invocation. If variadic macros are enabled, this special identifier can appear only in the replacement list of variadic macros.

Extended variadic macros (as implemented by certain pre-C99 compilers) use a slightly different syntax and allow the name of the variadic parameter to be chosen (instead of `.../__VA_ARGS__`):

```
#define EVM(x, args...) x(args)
void f() { EVM(sprintf, "%s %d\n", "Three args for ", 1); }
/* Same expansion as previous example. */
```

In addition, enabling extended variadic macros adds a special behavior to the token pasting operator `##` when it is followed by an empty or omitted macro argument: A preceding comma (possibly followed by white space) is erased. Hence,

```
#define EVM2(fmt, args) printf(fmt , ## args)
EVM2("Hello World\n")
```

expands to `printf("Hello World\n")` and the extraneous comma is erased.

- If the `LONG_LONG_ALLOWED` switch is `TRUE`,
 - the `long long` and `unsigned long long` types are accepted;
 - integer constants suffixed by `LL` are given the type `long long`, and those suffixed by `ULL` are given the type `unsigned long long` (any of the suffix letters may be written in lower case);
 - the specifier `%lld` is recognized in `printf` and `scanf` formatting strings; and
 - the `long long` types are accommodated in the usual arithmetic conversions.
- `restrict` may be used as a type qualifier for object pointer types and function parameter arrays that decay to pointers. Its presence is recorded in the `IL` so that back ends can perform optimizations that would otherwise be prevented because of possible aliasing. The keyword is enabled by `--restrict`; this is also available in C++.
- If `VLA_ALLOWED` is `TRUE`, variable length arrays (VLAs) are supported. VLA types may appear only in the declaration of an identifier that belongs to a block or function-prototype scope. For example:

```
void addscalar(int n, int m, double a[n][m], double x) {
    int i, j;
    for (i=0; i<n; i++)
        for (j=0; j<m; j++)
            a[i][j] += x;      /* a refers to a VLA with n*m elements. */
}
double A[4][8];
void f() {
    addscalar(4, 8, A, 3.14);
}
```

See also command line options `--vla` and `--no_vla`.

- When `DESIGNATED_INITIALIZER_ENABLING_POSSIBLE` is `TRUE`, designators may be accepted in initializers for aggregates. Designators are not allowed in C++ mode, however. The global variables `designators_allowed` and `extended_designators_allowed` are controlled by configuration flags and command line options. In turn they control which variety (if any) of designators is recognized. See also

command line options `--designators`, `--no_designators`, `--extended_designators` and `--no_extended_designators`.

When `designators_allowed` is `TRUE`, designators of the forms `.x` and `[k]` are accepted. They can be concatenated to reach nested aggregate elements. For example:

```
struct X { double a; int b[10] } x
        = { .b = { 1, [5] = 2 }, .b[3] = 1, .a = 42.0 };
```

In addition, when `extended_designators_allowed` is `TRUE`, designators of the form `x:` and `[m ... n]` are accepted and the assignment (`=`) token becomes optional after array element designators. Field designators of the form `x:` cannot immediately be followed by an assignment token (`=`) or another designator. Examples:

```
struct X { double a; int b[10] } x
        = { b: { 1, [5 ... 9] = 2 }, .b[7] 1, a: 42.0 };
struct Y y = { b:[3] /* Error */ = 7, a: = /* Error */ 42.0 };
```

Designators permit multiple initializations of the same subobject: only the last value is retained, but side-effects of prior initializing expressions do occur.

- Compound literals are supported in expressions. For example,

```
int *p = (int []){1, 2, 3};
```

creates an unnamed lvalue of type `int[3]` initialized as indicated. `p` is initialized to point to that array. Compound literals are not allowed in C++ mode.

- The `__generic` pseudo-macro is implemented. This is an EDG extension used to support the type-generic math header `<tgmath.h>`. The form of the macro reference is:

```
__generic(x, y, z, func_d, func_f, func_l, func_cd, func_cf, func_cl)
```

where `x` and the optional `y` and `z` are the arguments with which a type-generic function is called, and the remaining 6 arguments are the names of functions from which is selected the actual function to be called. The suffixes with which the function names are supplied here correspond to function parameter types of double, float, long double, double `_Complex`, float `_Complex`, and long double `_Complex`, respectively. The order is fixed. Function names may be omitted. For example, `<tgmath.h>` may have the following macros defined:

```
#define sin(x)    __generic(x,,, sin, sinf, sinl, csin, csinf, csinl)(x)
#define fmax(x,y) __generic(x, y,,fmax, fmaxf, fmaxl,,)(x, y)
#define conjg(x) __generic(x,,, ,,, conjg, conjgf, conjgl)(x)
```

Note that `sin` and `conjg` take only one argument, that `fmax` has no forms that accept complex arguments, and that `conjg` has no forms that accept real arguments.

There is also a `__genericfx` pseudo-macro that provides a similar capability for fixed-point types. The form of the macro reference is:

```
__genericfx(x, fnc_hr, fnc_uhr, fnc_r, fnc_ur, fnc_lr, fnc_ulr,
            fnc_hk, fnc_uhk, fnc_k, fnc_uk, fnc_lk, fnc_ulk)
```

where `x` is as above and the remaining 12 arguments are the names of functions, with the suffixes corresponding to function parameter types of short `_Fract`, unsigned short `_Fract`, `_Fract`, unsigned `_Fract`, long `_Fract`, unsigned long `_Fract`, short `_Accum`, unsigned short `_Accum`, `_Accum`, unsigned `_Accum`, long `_Accum`, unsigned long `_Accum`, respectively.

- The keywords `__I__`, `__NAN__`, and `__INFINITY__` are defined as, respectively, a float `_Imaginary` *i*, a float Not-a-Number, and a float positive Infinity. These are intended to be used by library writers to implement the C99 macros `I`, `NAN`, and `INFINITY`. `__NAN__` and `__INFINITY__` are also available in all modes, not just C99 mode.

1.9 C Dialect Accepted

- A trailing array member of unspecified length in a struct type is called a flexible array member. Such members are a standard feature in C99. In default C99 mode (but not in strict C99 mode) trailing struct members can also have a type containing a flexible array member. For example:

```
struct F {
    int i;
    int f[]; // OK in default and strict C99 modes
};
struct X {
    F x; // OK in default C99 mode (but not in strict mode)
};
```

1.9.2 ANSI C Extensions

The following extensions are accepted (these are flagged if the `-A` or `-a` option is specified):

- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.
- `__ALIGNOF__` (or `__alignof__`) is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It may be followed by a type or expression in parentheses:

```
__ALIGNOF__(type)           __alignof__(type)
__ALIGNOF__(expression)     __alignof__(expression)
```

The expression in the second form is not evaluated.

- `__INTADDR__(expression)` scans the enclosed expression as a constant expression, and converts it to an integer constant (it is used in the `offsetof` macro).
- A number of “type traits pseudo-functions” (taking one or two type names) are accepted: `__has_nothrow_assign`, `__has_nothrow_constructor`, `__has_nothrow_copy`, `__has_trivial_assign`, `__has_trivial_constructor`, `__has_trivial_copy`, `__has_trivial_destructor`, `__is_abstract`, `__is_base_of`, `__is_class`, `__is_convertible_to`, `__is_empty`, `__is_enum`, `__is_pod`, `__is_polymorphic`, `__is_standard_layout`, `__is_trivial`, and `__is_union`.

```
double x[__is_union(union U)]; // Okay.
```

These are silently accepted even in strict modes. They ease the implementation C++11 metaprogramming templates (most of which were first introduced by ISO/IEC TR 19768).

- Bit fields may have base types that are enums or integral types besides `int` and `unsigned int`. This matches A.6.5.8 in the ANSI Common Extensions appendix.
- If the `ADDR_OF_BIT_FIELD_ALLOWED` flag is `TRUE`, the address of a bit field may be taken if the bit field has the same size and alignment as one of the integral types. A warning is issued.
- The last member of a `struct` may have an incomplete array type. It may not be the only member of the struct (otherwise, the struct would have zero size).
- A file-scope array may have an incomplete `struct`, `union`, or `enum` type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not `extern`. In C++, an incomplete `class` is also allowed.
- Static functions may be declared in function and block scopes. Their declarations are moved to the file scope.

- enum tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the unsigned `int` range but not in the `int` range. A warning is issued for suspicious cases.

```
/* When ints are 32 bits: */
enum a {w = -2147483648}; /* No warning */
enum b {x = 0x80000000}; /* No warning */
enum c {y = 0x80000001}; /* No warning */
enum d {z = 2147483649}; /* Warning */
```

- An extra comma is allowed at the end of an enum list. A remark is issued except in `pcc` mode.
- The final semicolon preceding the closing `}` of a struct or union specifier may be omitted. A warning is issued except in `pcc` mode.
- A label definition may be immediately followed by a right brace. (Normally, a statement must follow a label definition.) A warning is issued.
- An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union need not be enclosed in braces. ANSI C requires the braces.
- In an initializer, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- The address of a variable with `register` storage class may be taken. A warning is issued.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- `long float` is accepted as a synonym for `double`.
- If `PTR_TO_INCOMP_ARRAY_ARITHMETIC_ALLOWED` is `TRUE`, pointers to incomplete arrays may be used in pointer addition, subtraction, and subscripting:

```
int (*p)[];
...
q = p[0];
```

A warning is issued if the value added or subtracted is anything other than a constant zero. Since the type pointed to by the pointer has zero size, the value added to or subtracted from the pointer is multiplied by zero and therefore has no effect on the result.

- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as the same type. A warning is issued.
- Dollar signs can be accepted in identifiers through use of a command line option or by setting a configuration parameter. The default is to not allow dollar signs in identifiers.
- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token. (If the `-A` or `-a` option is specified, of course, the `pp-number` syntax is used.)
- Assignment and pointer difference are allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to same-sized integral types (e.g.,

1.9 C Dialect Accepted

typically, `int *` and `long *`). A warning is issued except in `pcc` mode. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.

- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (e.g., `int **` to `const int **`). Comparisons and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, some operators allow such things, and others (generally, where it does not make sense) do not allow them.
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. A warning is issued. This extension is not allowed in C++ mode.
- A pointer to `void` may be implicitly converted to or from a pointer to a function type.
- If the `ATT_PREPROCESSING_EXTENSIONS_ALLOWED` switch is `TRUE`, the `#assert` preprocessing extensions of AT&T System V release 4 are allowed. These allow definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. A predicate name is given a definition by a preprocessing directive of the form

```
#assert name
#assert name(token-sequence)
```

which defines the predicate *name*. In the first form, the predicate is not given a value. In the second form, it is given the value *token-sequence*.

Such a predicate can be tested in a `#if` expression, as follows

```
#name(token-sequence)
```

which has the value 1 if a `#assert` of that *name* with that *token-sequence* has appeared, and 0 otherwise. A given predicate may be given more than one value at a given time.

A predicate may be deleted by a preprocessing directive of the form

```
#unassert name
#unassert name(token-sequence)
```

The first form removes all definitions of the indicated predicate name; the second form removes just the indicated definition, leaving any others there may be.

- GNU line directive flags are accepted on `#line` directives. The system header flag is recognized, and the associated code is treated as if found in a system include directory.
- `asm` statements and declarations are accepted. This is disabled in strict ANSI C mode (`-A` or `-a` and `-m` options) since it conflicts with the ANSI C standard for something like

```
asm("xyz");
```

which ANSI C interprets as a call of an implicitly-defined function `asm` and which by default the front end interprets as an `asm` statement.

- When `ASM_FUNCTION_ALLOWED` is `TRUE`, `asm` functions are accepted, and `__asm` is recognized as a synonym for `asm`. An `asm` function body is represented in the IL by an uninterpreted null-terminated string containing the text that appears in the source (including the text of source comments when `INCLUDE_COMMENTS_IN_ASM_FUNC_BODY` is `TRUE`). An `asm` function must be declared with no storage class, with a prototyped parameter list, and with no omitted parameters:

```
asm void f(int,int) {
    ...
}
```

If `asm` functions are recognized and the C-generating back end is used, it is required that ANSI-C be generated, not K&R C, because the `asm` function must be put out with a prototyped parameter list.

- If `ALLOW_NONSTANDARD_ANONYMOUS_UNIONS` is `TRUE`, an extension is supported to allow constructs similar to C++ anonymous unions, including the following:
 - not only anonymous unions but also anonymous structs are allowed — that is, their members are promoted to the scope of the containing struct and looked up like ordinary members;
 - they can be introduced into the containing struct by a `typedef` name — they needn't be declared directly, as with true anonymous unions; and
 - a tag may be declared (C mode only).

Among the restrictions: the extension only applies to constructs within structs.

- If `DEFAULT_ADDRESS_OF_ELLIPSIS_ALLOWED` is `TRUE`, the expression `& . . .` is accepted in the body of a function in which an ellipsis appears in the parameter list. It is needed to support some versions of macro `va_start` in `stdarg.h`.
- If `DEFAULT_ALLOW_ELLIPSIS_ONLY_PARAM_IN_C_MODE` is `TRUE`, an ellipsis may appear by itself in the parameter list of a function declaration — e.g., `f(. . .)`. A diagnostic is issued in strict ANSI C mode.
- External entities declared in other scopes are visible. A warning is issued.

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */ }
```

- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.
- The nonstandard preprocessing directive `#include_next` is supported. This is a variant of the `#include` directive. It searches for the named file only in the directories on the search path that follow the directory in which the current source file (the one containing the `#include_next` directive) is found. (This is an extension found in the GNU C compiler.)
- The nonstandard preprocessing directive `#warning` is supported. It is similar to the `#error` directive, but results in a warning instead of a catastrophic error when processed. This directive is not recognized in strict mode. (This is an extension found in GNU C compilers.)

In the following areas considered “undefined behavior” by the ANSI C standard, the front end does the following:

- Adjacent wide and non-wide string literals are not concatenated unless `wchar_t` and `char` are the same type. (In C++ mode, when `wchar_t` is a keyword, adjacent wide and non-wide string literals are never concatenated.)
- In character and string escapes, if the character following the `\` has no special meaning, the value of the escape is the character itself. Thus `"\s" == "s"`. A warning is issued.
- A `struct` that has no named fields but at least one unnamed field is accepted by default, but a diagnostic (a warning or error) is issued in strict ANSI C mode.

1.9.3 K&R/pcc Mode

When `pcc` mode is specified, the front end accepts the traditional C language defined by *The C Programming Language*, first edition, by Kernighan and Ritchie (K&R), Prentice-Hall, 1978. In addition, it provides almost complete

1.9 C Dialect Accepted

compatibility with the Reiser `cpp` and Johnson `pcc` widely used as part of UNIX systems; since there is no documentation of the exact behavior of those programs, complete compatibility cannot be guaranteed.

In general, when compiling in `pcc` mode, the front end attempts to interpret a source program that is valid to `pcc` in the same way that `pcc` would. However, ANSI features that do not conflict with this behavior are not disabled.

In some cases where `pcc` allows a highly questionable construct, the front end will accept it but give a warning, where `pcc` would be silent (for example: `0x`, a degenerate hexadecimal number, is accepted as zero).

The known cases where the front end is not compatible with `pcc` are the following:

- Token pasting is not done outside of macro expansions (i.e., in the primary source line) when two tokens are separated only by a comment. That is, `a/**/b` is not considered to be `ab`. The `pcc` behavior in that case can be gotten by preprocessing to a text file and then compiling that file.

The textual output from preprocessing is also equivalent but not identical: the blank lines and white space will not be exactly the same.

- `pcc` will consider the result of a `?:` operator to be an lvalue if the first operand is constant and the second and third operands are compatible lvalues. This front end will not.
- `pcc` mis-parses the third operand of a `?:` operator in a way that some programs exploit:

```
i ? j : k += 1
```

is parsed by `pcc` as

```
i ? j : (k += 1)
```

which is not correct, since the precedence of `+=` is lower than the precedence of `?:`. This front end will generate an error for that case.

- `lint` recognizes the keywords for its special comments anywhere in a comment, regardless of whether or not they are preceded by other text in the comment. The front end only recognizes the keywords when they are the first identifier following an optional initial series of blanks and/or horizontal tabs. `lint` also recognizes only a single digit of the `VARARGS` count; the front end will accumulate as many digits as appear.

The differences in `pcc` mode relative to the default ANSI mode are as follows:

- The keywords `signed`, `const`, and `volatile` are disabled, to avoid problems with items declared with those names in old-style code. Those keywords were ANSI C inventions. The other non-K&R keywords (`enum` and `void`) are judged to have existed already in code and are not disabled.
- Declarations of the form

```
typedef some-type void;
```

are ignored.
- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form `p->field` is allowed even if `p` does not point to a `struct` or union that contains `field`. `p` must be a pointer or an integer. Likewise, `x.field` is allowed even if `x` is not a `struct` or union that contains `field`. `x` must be an lvalue. For both cases, all definitions of `field` as a field must have the same offset within their `struct` or union.
- If the configuration flag `TARG_NO_ERROR_ON_INTEGER_OVERFLOW` is `TRUE`, overflows detected while folding signed integer operations on constants will cause warnings rather than errors. Usually this should be set to match the desired target machine behavior on integer operations in C.

- If the configuration flag `SAME_REPR_INTS_INTERCHANGEABLE_IN_IL` is `TRUE`, integral types with the same representation (size, signedness, and alignment) will be considered identical and may be used interchangeably. For example, this means that `int` and `long` will be interchangeable if they have the same size.
- A warning will be issued for a `&` applied to an array. The type of such an operation is “address of array element” rather than “address of array”.
- For the shift operators `<<` and `>>`, the usual arithmetic conversions are done, the right operand is converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands, and the result type is the type of the left operand. The effect of this difference is that, in `pcc` mode, a `long` shift count will force the shift to be done as `long`.
- When preprocessing output is generated, the line-identifying directives will have the `pcc` form instead of the ANSI form.
- String literals will not be shared. Identical string literals will cause multiple copies of the string to be allocated.
- `sizeof` may be applied to bit fields; the size is that of the underlying type (e.g., `unsigned int`).
- `lvalues` cast to a type of the same size remain `lvalues`, except when they involve a floating-point conversion.
- When a function parameter list begins with a `typedef` identifier, the parameter list is considered prototyped only if the `typedef` identifier is followed by something other than a comma or right parenthesis:

```
typedef int t;
int f(t) {}      /* Old-style list */
int g(t x) {}   /* Prototyped list, parameter x of type t */
```

That is, function parameters are allowed to have the same names as `typedefs`. In the normal ANSI mode, of course, any parameter list that begins with a `typedef` identifier is considered prototyped, so the first example above would give an error.

- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, used, and never defined is treated as if its storage class were `extern`.
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern` (in ANSI mode, the number of elements is changed to 1, and the storage class remains unspecified).
- The empty declaration

```
struct x;
```

will not hide an outer-scope declaration of the same tag.
- In a declaration of a member of a `struct` or `union`, no diagnostic is issued for omitting the declarator list; nevertheless, such a declaration has no effect on the layout. For example,

```
struct s {char a; int; char b[2];} v;      /* sizeof(v) is 3 */
```
- `enums` are always given type `int`. In ANSI mode, and depending on the configuration flag `enum_types_can_be_smaller_than_int`, smaller integral types will be used if possible.
- No warning is generated for a storage specifier appearing in other than the first position in a list of specifiers (as in `int static`).
- `short`, `long`, and `unsigned` are treated as “adjectives” in type specifiers, and they may be used to modify a `typedef` type.

1.9 C Dialect Accepted

- A “plain” `char` is considered to be the same as either `signed char` or `unsigned char`, depending on the installation default and command-line options. In ANSI C, “plain” `char` is a third type distinct from both `signed char` and `unsigned char`.
- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- `float` function parameters are promoted to `double` function parameters.
- `float` functions are promoted to `double` functions.
- Declaration specifiers are allowed to be completely omitted in declarations (ANSI C allows this only for function declarations). Thus

```
    i;
```

declares `i` as an `int` variable. A warning is issued.
- The `=` preceding an initializer may be omitted. A warning is issued. See K&R first edition, Appendix A, section 17 (anachronisms). This is accepted only if `C_ANACHRONISMS_ALLOWED` is `TRUE`.
- All `float` operations are done as `double`.
- `__STDC__` is left undefined.
- Extra spaces to prevent pasting of adjacent confusable tokens are not generated in textual preprocessing output.
- The first directory searched for include files is the directory containing the file containing the `#include` instead of the directory containing the primary source file.
- Trigraphs are not recognized.
- Comments are deleted entirely (instead of being replaced by one space) in preprocessing output.
- `0x` is accepted as a hexadecimal 0, with a warning.
- `1E+` is accepted as a floating-point constant with an exponent of 0, with a warning.
- The compound assignment operators may be written as two tokens (e.g., `+=` may be written `+ =`).
- The compound assignment operators may be written in their old-fashioned reversed forms (e.g., `--` may be written `=-`). A warning is issued. This is described in K&R first edition, Appendix A, section 17 (anachronisms). This is accepted only if `C_ANACHRONISMS_ALLOWED` is `TRUE`.
- The digits 8 and 9 are allowed in octal constants.
- A warning rather than an error is issued for integer constants that are larger than can be accommodated in an `unsigned long`. The value is truncated to an acceptable number of low-order bits.
- The types of large integer constants are determined according to the K&R rules (they won’t be `unsigned` in some cases where ANSI C would define them that way). Integer constants with apparent values larger than `LONG_MAX` are typed as `long` and are also marked as “non-arithmetic”, which suppresses some warnings when using them.
- The escape `\a` (alert) is not recognized in character and string constants.
- Macro expansion is done differently. Arguments to macros are not macro-expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is possible and is checked for.

- Token pasting inside macro expansions is done differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- Macro parameter names inside character and string constants are recognized and substituted for.
- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- Extra `#elses` (after the first has appeared in an `#if` block) are ignored, with a warning.
- Expressions in a `switch` statement are cast to `int`; this differs from the ANSI C definition in that a `long` expression is (possibly) truncated.
- The promotion rules for integers are different: `unsigned char` and `unsigned short` are promoted to `unsigned int`.
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

1.9.4 Extensions Accepted in SVR4 Compatibility Mode

The following extensions are accepted in SVR4 C compatibility mode:

- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- The sequence `/**/` in a macro definition is treated as equivalent to the token-pasting operator `##`.
- `l`values cast to a type of the same size remain `l`values, except when they involve a floating-point conversion.
- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form `p->field` is allowed even if `p` does not point to a `struct` or `union` that contains `field`. `p` must be a pointer. Likewise, `x.field` is allowed even if `x` is not a `struct` or `union` that contains `field`. `x` must be an `l`value. For both cases, all definitions of `field` as a field must have the same offset within their `struct` or `union`.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- Incompatible external object declarations are allowed if the object types share the same underlying representation.
- Certain incompatible function declarations are allowed. A warning is issued.

```
typedef unsigned int size_t;
extern size_t strlen(const char *);
extern int strlen(); /* Warning */
```

1.10 Namespace Support

Namespaces are enabled by default (see `DEFAULT_NAMESPACES_ENABLED`) except in the cfront modes. The command-line options `--namespaces` and `--no_namespaces` can be used to enable or disable the features.

When doing name lookup in a template instantiation, some names must be found in the context of the template definition while others may also be found in the context of the template instantiation. The front end implements two different instantiation lookup algorithms: the one mandated by the standard (referred to as “dependent name lookup”), and the one that existed before dependent name lookup was implemented.

Dependent name lookup is done in strict mode (unless explicitly disabled by another command-line option) or when dependent name processing is enabled by either a configuration flag or command-line option.

Dependent Name Processing

When doing dependent name lookup, the front end implements the instantiation name lookup rules specified in the standard. This processing requires that nonclass prototype instantiations be done. This in turn requires that the code be written using the `typename` and `template` keywords as required by the standard.

Lookup Using the Referencing Context

When not using dependent name lookup, the front end uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but does so in such a way that is more compatible with existing code and existing compilers.

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 that includes both names from the context of the template definition and names from the context of the instantiation. Here’s 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.

For details of the algorithm implemented, see the Symbol Table chapter (in particular the section entitled “Instantiation Context Lookup”).

Argument Dependent Lookup

When argument-dependent lookup is enabled, functions made visible using argument-dependent lookup overload with those made visible by normal lookup. The standard requires that this overloading occur even when the name found by normal lookup is a block `extern` declaration. The front end does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block `extern`.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup, even if the program makes no use of namespaces. For example:

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

Runtime Support

The headers and runtime provided with the EDG front end support two different versions (i.e., with and without namespaces) under control of `__EDG_RUNTIME_USES_NAMESPACES`, a preprocessing option defined by the front end.

1.11 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

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*.

1.11 Template Instantiation

`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 modern C++ language requires that a specialization be declared in every compilation in which it is used, but for compatibility with existing code and older compilers the EDG front end does not require that in some modes. See the command-line option `--no_distinct_template_signatures`.)

- C++ templates can be *exported* (i.e., declared with the keyword `export`). Such templates can be used in a translation unit that does not contain the definition of the template to instantiate. The instantiation of such a template must be delayed until the template definition has been found.
- 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.)

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.

The EDG C++ front end provides an instantiation mechanism that does automatic instantiation at link time. For cases where the programmer wants more explicit control over instantiation, the front end also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process.

1.11.1 Automatic Instantiation

The goal of an automatic instantiation mode is to provide painless instantiation. The programmer 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/SCO'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.

The programmer using *cfront* must follow a particular coding convention: all templates must be declared in ".h" files, and for each such file there must be a corresponding ".C" file containing the associated definitions. The compiler is never told about the ".C" 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" file name and replacing its suffix.¹

¹ The actual implementation allows for several different suffixes and provides a command-line option to change the suffixes sought.

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, *cfront* 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.

The programmer using Borland’s compiler 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 “.C” (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.

EDG’s approach is a little different. It requires that, for each instantiation of a non-exported template, 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.¹ 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 EDG’s approach. See the section on implicit inclusion.
- The ad hoc approach: the programmer makes sure that the files that define template entities also have the definitions of all the available types, and adds code or pragmas in those files to request instantiation of the entities there.

Exported templates are also supported by the EDG automatic instantiation method, but they require additional mechanisms explained further on.

The EDG automatic instantiation method works as follows for non-exported templates:²

¹ 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.

² It should be noted that automatic instantiation, more than most aspects of the C++ language, requires environmental support outside of the compiler. This is likely to be operating-system and object-format dependent. The front end is delivered with an implementation that works under certain UNIX systems, and that involves code in the front end proper, in IL lowering, in a separate prelinker program, and in the `eccpp` driver script. Because the environmental issues vary so much from system to system, this code should be viewed as a sample implementation that may need adaptation to work on other systems.

1.11 Template Instantiation

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files (with, by default, a “.ti” suffix) are generated and contain information about things that *could* have been instantiated in each compilation¹.
2. When the object files are linked together, a program called the *prelinker* 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. 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 an 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 instantiation request 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 instantiation request file for that file and obeys the 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). The compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during the compilation the compiler has the opportunity to instantiate a referenced entity that is not on that list, it goes ahead and does the instantiation. It passes back to the prelinker (in the definition list file) a list of the instantiations that it has “adopted” in this way, so the prelinker can assign them to the file. This adoption process allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.
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 instantiation request files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the instantiation request 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 the programmer provides 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 the programmer adds 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 instantiation request file.

The instantiation request 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,

¹ As a configuration option, and in older versions of the front end, the information about entities that could be instantiated, etc. can be represented in the names of generated variables added to the object program. For example, a name `__CBI__mangled-name` indicates that the entity with the given mangled name could be instantiated, but wasn’t. This alternate scheme has the advantage that the object file and the associated instantiation flags can never be out of sync. It has the disadvantage that the object files produced can be quite a bit larger because of the name strings for these generated external variables.

the instantiation request file for the file getting the errors must be deleted manually to allow the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, it will issue a message like

```
C++ prelinker: A<int>::f() assigned to file test.o
C++ prelinker: executing: /edg/bin/eccp -c test.c
```

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer 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 can be configured out altogether by setting the configuration flag `AUTOMATIC_TEMPLATE_INSTANTIATION` to `FALSE`. It can be turned on or off using the `--auto_instantiation` and `--no_auto_instantiation` command-line options. If automatic instantiation is turned off, the template information file is not generated.

1.11.2 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:

<code>-tnone</code>	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.
<code>-tused</code>	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.
<code>-tall</code>	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. Nonmember template functions will be instantiated even if the only reference was a declaration.
<code>-tlocal</code>	Similar to <code>-tused</code> 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. <code>-tlocal</code> can not be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it will be disabled by the <code>-tlocal</code> option. If automatic instantiation is not enabled by default, use of <code>-tlocal</code> and <code>-T</code> is an error.

In the case where the `eccp` script is given a single file to compile and link, e.g.,

```
eccp t.c
```

1.11 Template Instantiation

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

1.11.3 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.¹

The argument to the instantiation pragma may be:

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

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;
```

¹ At the moment, the `can_instantiate` pragma ends up forcing the instantiation of the template instance even if it isn't referenced somewhere else in the program; that's a weakness of the initial implementation which we expect to address.

```
    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.

1.11.4 Implicit Inclusion

When implicit inclusion is enabled, the front end 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 “.C” 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.C` 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 front end needs to know the path name specified in the original include 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 front end will not attempt implicit inclusion for source code containing `#line` directives.

The file to be implicitly included is found by replacing the file suffix with each of the suffixes specified in the instantiation file suffix list. The normal include search path mechanism is then used to look for the file to be implicitly included.

`DEFAULT_INSTANTIATION_FILE_SUFFIX_LIST` defines the set of definition-file suffixes tried. By default, the list is “.c”, “.C”, “.cpp”, “.CPP”, “.cxx”, “.CXX”, and “.cc”.

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 configured out altogether by setting the configuration flag `INSTANTIATION_BY_IMPLICIT_INCLUSION` to `FALSE`. It can be turned on or off using the `--implicit_include` and `--no_implicit_include` command-line options.

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

1.11 Template Instantiation

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.

1.11.5 Exported Templates

Exported templates are templates declared with the keyword `export`. Exporting a class template is equivalent to exporting each of its static data members and each of its non-inline member functions. An exported template is special because its definition does not need to be present in a translation unit that uses that template. In other words, the definition of an exported (non-class) template does not need to be explicitly or implicitly included in a translation unit that instantiates that template. For example, the following is a valid C++ program consisting of two separate translation units:

```
// File 1:
#include <stdio.h>
static void trace() { printf("File 1\n"); }

export template<class T> T const& min(T const&, T const&);
int main() {
    trace();
    return min(2, 3);
}

// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

export template<class T> T const& min(T const &a, T const &b) {
    trace();
    return a<b? a: b;
}
```

Note that these two files are separate translation units: one is not included in the other. That allows the two functions `trace()` to coexist (with internal linkage).

Support for exported templates can be enabled using the `--export` command-line option (it is enabled by default in strict ANSI mode; `-A` or `--strict`). For example, the program above could be built as follows using the `eccp` script:

```
eccp --export -c file_1.c
eccp --export -c file_2.c
eccp file_1.o file_2.o
```

1.11.5.1 Finding the exported template definition

The automatic instantiation of exported templates is somewhat similar (from a user's perspective) to that of regular (included) templates. However, an instantiation of an exported template involves at least two translation units: one which requires the instantiation, and one which contains the template definition.

When a file containing definitions of exported templates is compiled, a file with a `.et` suffix is created and some extra information is included in the associated `.ti` file. The `.et` files are used later by the front end to find the translation unit that defines a given exported template.

When a file that potentially makes use of exported templates is compiled, the compiler must be told where to look for “.et” files for exported templates used by a given translation unit. By default, the compiler looks in the current directory. Other directories may be specified with the `--template_directory` option. Strictly speaking, the “.et” files are only really needed when it comes time to generate an instantiation. This means that code using exported templates can be compiled without having the definitions of those templates available. Those definitions must be available by the time prelinking is done (or when explicit instantiation is done).

The “.et” files only inform the front end about the location of exported template definitions; they do not actually contain those definitions. The sources containing the exported template definitions must therefore be made available at the time of instantiation (usually, when prelinking is done). In particular, the export facility is *not* a mechanism for avoiding the publication of template definitions in source form.

1.11.5.2 Secondary translation units

An instantiation of an exported template can be triggered by the prelinker, by an explicit instantiation directive, or by the `-tused` command-line option. In each case, the translation unit that contains the initial point of instantiation will be processed as the *primary translation unit*. Based on information it finds in the “.et” files, the front end will then load and parse the translation unit containing the definition of the template to instantiate. This is a *secondary translation unit*. The simultaneous processing of the primary and secondary translation units enables the front end to create instantiations of the exported templates (which can include entities from both translation units). This process may reveal the need for additional instantiations of exported templates, which in turn can cause additional secondary translation units to be loaded¹.

When secondary translation units are processed, the declarations they contain are checked for consistency. This process may report errors that would otherwise not be caught. Many these errors are so-called “ODR violations” (ODR stands for “one-definition rule”). For example:

```
// File 1:
struct X {
    int x;
};

int main() {
    return min(2, 3);
}

// File 2:
struct X {
    unsigned x; // Error: X::x declared differently in File 1
};

export template<class T> T const& min(T const &a, T const &b) {
    return a<b? a: b;
}
```

If there are no errors, the instantiations are generated in the output associated with the primary translation unit (or in separate associated files in one-instantiation-per-object mode). This may also require that entities with internal

¹ As a consequence, using exported templates may require considerably more memory than similar uses of regular (included) templates.

1.11 Template Instantiation

linkage in secondary translation units be “externalized” so they can be accessed from the instantiations in the primary translation unit.

1.11.5.3 Libraries with exported templates

Typically a (non-export) library consists of an *include* directory and a *lib* directory. The include directory contains the header files required by users of the library and the lib directory contains the object code libraries that client programs must use when linking programs.

With exported templates, users of the library must also have access to the source code of the exported templates and the information contained in the associated “.et” files. This information should be placed in a directory that is distributed along with the include and lib directories: This is the *export* directory. It must be specified using the `--template_directory` option when compiling client programs.

The recommended procedure to build the export directory is as follows:

1. For each “.et” file in the original source directory, copy the associated source file to the export directory.
2. Concatenate all of the “.et” files into a single “.et” file (e.g., `mylib.et`) in the export directory. The individual “.et” files could be copied to the export directory, but having all of the “.et” information in one file will make use of the library more efficient.
3. Create an `export_info` file in the export directory. The `export_info` file specifies the include search paths to be used when recompiling files in the export directory. If no `export_info` file is provided, the include search path used when compiling the client program that uses the library will also be used to recompile the library exported template files.

The `export_info` file consists of a series of lines of the form

```
include=x
```

or

```
sys_include=x
```

where `x` is a path name to be placed on the include search path. The directories are searched in the order in which they are encountered in the `export_info` file. The file can also contain comments—which begin with a “#”—and blank lines. Spaces are ignored but tabs are not currently permitted. For example:

```
# The include directories to be used for the xyz library

include = /disk1/xyz/include
sys_include = /disk2/abc/include
include=/disk3/jkl/include
```

The include search path specified for a client program is ignored by the front end when it processes the source in the export library, except when no `export_info` file is provided. Command-line macro definitions specified for a client program are also ignored by the front end when processing a source file from the export library; the command-line macros specified when the corresponding “.et” file was produced do apply. All other compilation options (other than the include search path and command-line macro definitions) used when recompiling the exported templates will be used to compile the client program.

When a library is installed on a new system, it is likely that the `export_info` file will need to be adapted to reflect the location of the required headers on that system.

1.12 Extern Inline Functions

Depending on the way in which the front end is configured, out-of-line copies of extern inline functions are either implemented using static functions, or are instantiated using a mechanism like the template instantiation mechanism. Note that out-of-line copies of inline functions are only required in cases where the function cannot be inlined, or when the address of the function is taken (whether explicitly by the user, by implicitly generated functions, or by compiler-generated data structures such as virtual function tables or exception handling tables).

When static functions are used, local static variables of the functions are promoted to global variables with specially encoded names, so that even though there may be multiple copies of the code, there is only one copy of such global variables. This mechanism does not strictly conform to the standard because the address of an extern inline function is not constant across translation units.

When the instantiation mechanism is used, the address of an extern inline function is constant across translation units, but at the cost of requiring the use of one of the template instantiation mechanisms, even for programs that don't use templates. Definitions of extern inline functions can be provided either through use of the automatic instantiation mechanism or by use of the `-tused` or `-tall` instantiation modes. There is no mechanism to manually control the definition of extern inline function bodies.

1.13 Predefined Macros

The front end 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. In addition, a table of predefined macro definitions may be processed when the compiler is invoked. This may be used for any kind of predefined macro, but in particular is used to define the macros that gcc and g++ define when using gcc and g++ modes. The format of the file and other information is described in section 18.16, "Predefined Macro Definition File," on page 454.

<code>__STDC__</code>	Defined in ANSI C mode and in C++ mode. In C++ mode the value may be redefined. Not defined in Microsoft compatibility mode.
<code>__DATE__</code>	Defined in all modes to the date of the compilation in the form "Mmm dd yyyy".
<code>__TIME__</code>	Defined in all modes to the time of the compilation in the form "hh:mm:ss".
<code>__FILE__</code>	Defined in all modes to the name of the current source file.
<code>__LINE__</code>	Defined in all modes to the current source line number within the current source file.
<code>__cplusplus</code>	Defined in C++ mode.
<code>c_plusplus</code>	Defined in cfront mode.
<code>__cplusplus_cli</code>	Defined in C++/CLI mode with the value 200406L.
<code>__STDC_VERSION__</code>	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 C89 Standard. In C99 mode, defined with the value 199901L.
<code>__STDC_HOSTED__</code>	Defined in C99 mode with the value specified by the <code>STDC_HOSTED</code> configuration macro.

1.13 Predefined Macros

<code>__STDC_IEC_559__</code>	Defined in C99 mode if the configuration macro <code>STDC_IEC_559</code> is TRUE.
<code>__STDC_IEC_559_COMPLEX__</code>	Defined in C99 mode if the configuration macro <code>STDC_IEC_559_COMPLEX</code> is TRUE.
<code>__STDC_ISO_10646__</code>	Defined in C99 mode if the configuration macro <code>STDC_ISO_10646</code> is TRUE. The value is specified by the <code>STDC_ISO_10646_VALUE</code> macro.
<code>__STDC_MB_MIGHT_NEQ_WC__</code>	Defined in C99 mode if the configuration macro <code>STDC_MB_MIGHT_NEQ_WC</code> is TRUE.
<code>__SIGNED_CHARS__</code>	Defined when plain <code>char</code> is signed. This is used in the <code><limits.h></code> header file to get the proper definitions of <code>CHAR_MAX</code> and <code>CHAR_MIN</code> .
<code>__FUNCTION__</code>	Defined in all modes to the name of the current function. An error is issued if it is used outside of a function. An EDG, GNU, and Microsoft extension. (Depending on the current mode, this identifier behaves as a variable or as a string literal; never as a preprocessor macro.)
<code>__func__</code>	Same as <code>__FUNCTION__</code> in C99, C++11, and GNU modes. (This identifier behaves as a variable; not as a preprocessor macro.)
<code>__PRETTY_FUNCTION__</code>	Defined in all modes to the name of the current function. This includes the return type and parameter types of the function. An error is issued if it is used outside of a function. An EDG and GNU extension. (Depending on the current mode, this identifier behaves as a variable or as a string literal; never as a preprocessor macro.)
<code>__FUNCSIG__</code>	Same as <code>__PRETTY_FUNCTION__</code> in Microsoft mode. (This identifier behaves as a string literal; not as a preprocessor macro.)
<code>__FUNCDNAME__</code>	Defined to the “decorated” name of the current function (i.e., its mangled name) in Microsoft mode. (This identifier behaves as a string literal; not as a preprocessor macro.)
<code>__BASE_FILE__</code>	Defined in all modes. Similar to <code>__FILE__</code> but indicates the primary source file rather than the current one (i.e., when the current file is an included file).
<code>_WCHAR_T</code>	Defined in C++ mode when <code>wchar_t</code> is a keyword. The name of this predefined macro is specified by a configuration flag. <code>_WCHAR_T</code> is the default.
<code>_WCHAR_T_DEFINED</code>	Defined in Microsoft C++ mode when <code>wchar_t</code> is a keyword.
<code>_NATIVE_WCHAR_T_DEFINED</code>	Defined in Microsoft C++ mode when <code>wchar_t</code> is a keyword (Microsoft version 1300 and above).
<code>_BOOL</code>	Defined in C++ mode when <code>bool</code> is a keyword. The name of this predefined macro is specified by a configuration flag. <code>_BOOL</code> is the default.
<code>__BOOL_DEFINED</code>	Defined in Microsoft C++ mode when <code>bool</code> is a keyword.
<code>__ARRAY_OPERATORS</code>	Defined in C++ mode when <code>array new</code> and <code>delete</code> are enabled. The name of this predefined macro is specified by a configuration flag. <code>__ARRAY_OPERATORS</code> is the default.

<code>__EXCEPTIONS</code>	Defined in C++ mode when exception handling is enabled. The name of this predefined macro is specified by a configuration flag. <code>__EXCEPTIONS</code> is the default.
<code>__RTTI</code>	Defined in C++ mode when RTTI is enabled. The name of this predefined macro is specified by a configuration flag. <code>__RTTI</code> is the default.
<code>__PLACEMENT_DELETE</code>	Defined in C++ mode when placement delete is enabled. The name of this predefined macro is specified by a configuration flag. <code>__PLACEMENT_DELETE</code> is the default. Placement delete is enabled when <code>ABI_CHANGES_FOR_PLACEMENT_DELETE</code> is TRUE and exception handling is enabled.
<code>__NO_LONG_LONG</code>	Defined in all modes when the <code>long long</code> type is not supported. The name of this predefined macro is specified by a configuration flag. <code>__NO_LONG_LONG</code> is the default.
<code>__EDG_RUNTIME_USES_NAMESPACES</code>	Defined in C++ mode when the configuration flag <code>RUNTIME_USES_NAMESPACES</code> is TRUE. The name of this predefined macro is specified by a configuration flag. <code>__EDG_RUNTIME_USES_NAMESPACES</code> is the default.
<code>__EDG_IMPLICIT_USING_STD</code>	Defined in C++ mode when the configuration flag <code>RUNTIME_USES_NAMESPACES</code> is TRUE and when <code>implicit_using_std</code> has been set to TRUE, either by a command line option or by the configuration flag <code>DEFAULT_IMPLICIT_USING_STD</code> , indicating that the standard header files should implicitly do a using-directive on the <code>std</code> namespace. The name of this predefined macro is specified by a configuration flag. <code>__EDG_IMPLICIT_USING_STD</code> is the default.
<code>__EDG_TYPE_TRAITS_ENABLED</code>	Defined in non-Microsoft modes when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., “ <code>__is_union</code> ”) are enabled. The name of this predefined macro is specified by a configuration flag. <code>__EDG_TYPE_TRAITS_ENABLED</code> is the default.
<code>__EDG__</code>	Always defined.
<code>__EDG_VERSION__</code>	Defined to an integral value that represents the version number of the front end. For example, version 2.30 is represented as 230.
<code>__EDG_IA64_ABI</code>	Defined as 1 when using a front end configured to use the IA-64 ABI.
<code>__embedded_cplusplus</code>	Defined as 1 in Embedded C++ mode.
<code>_MSC_VER</code>	Defined in Microsoft mode to the version number of the Microsoft compiler that is to be emulated. For example, 1100 is the value that corresponds to Visual C++ version 5.0.
<code>_MSC_EXTENSIONS</code>	Defined in Microsoft mode.
<code>_M_IX86</code>	Defined in Microsoft mode when the front end is built with the <code>_M_IX86</code> macro defined. It is defined to the same value used when the front end was built.
<code>_WIN32</code>	Defined in Microsoft mode.

1.13 Predefined Macros

<code>__INTEGRAL_MAX_BITS</code>	Defined in Microsoft mode to the number of bits in the largest integral type that is supported. The value depends on the way in which the front end was configured.
<code>__CPPRTTI</code>	Defined in Microsoft mode C++ when RTTI is enabled.
<code>__GNUC__</code>	Defined in GNU mode to the value specified by the <code>GCC_VERSION</code> configuration macro.
<code>__GNUC_MINOR__</code>	Defined in GNU mode to the value specified by the <code>GCC_MINOR_VERSION</code> configuration macro.
<code>__VERSION__</code>	Defined in GNU mode to the value specified by the <code>GCC_VERSION_STRING</code> configuration macro.
<code>__PRAGMA_REDEFINE_EXTNAME</code>	Defined if all modes when the configuration flag <code>REDEFINE_EXTNAME_PRAGMA_ENABLED</code> is TRUE.
<code>__upc__</code>	Defined when UPC extensions are enabled.
<code>__COUNTER__</code>	Defined in Microsoft mode. Returns an integer value, starting with zero and incremented each time the macro is used.
<code>__TIMESTAMP__</code>	Defined in Microsoft mode. Returns a string literal containing the modification date and time of the source file in which it is specified.
<code>__EDG_IA64_ABI</code>	Defined in C++ mode when using the IA-64 ABI. The name of this predefined macro is specified by a configuration flag. <code>__EDG_IA64_ABI</code> is the default.
<code>__EDG_IA64_ABI_USE_INT_STATIC_INIT_GUARD</code>	Defined in C++ mode when using the IA-64 ABI when guard variables are <code>int-sized</code> (i.e., in the ARM EABI). The name of this predefined macro is specified by a configuration flag. <code>__EDG_IA64_ABI_USE_INT_STATIC_INIT_GUARD</code> is the default.
<code>__EDG_IA64_ABI_VARIANT_CTORS_AND_DTORS_RETURN_THIS</code>	Defined in C++ mode when using the IA-64 ABI when constructors and destructors return the <code>this</code> pointer (i.e., in the ARM EABI). The name of this predefined macro is specified by a configuration flag. <code>__EDG_IA64_ABI_VARIANT_CTORS_AND_DTORS_RETURN_THIS</code> is the default.
<code>__EDG_SIZE_TYPE__</code>	Defined to be the type of <code>size_t</code> .
<code>__EDG_PTRDIFF_TYPE__</code>	Defined to be the type of <code>ptrdiff_t</code> .
<code>__CHAR16_T_AND_CHAR32_T</code>	Defined in C++ mode when <code>char16_t</code> and <code>char32_t</code> are keywords. The name of this predefined macro is specified by a configuration macro. <code>__CHAR16_T_AND_CHAR32_T</code> is the default.
<code>__CHAR16_TYPE__</code>	Defined in GNU mode as the underlying type for <code>char16_t</code> (GNU version 40400 and above).

`__CHAR32_TYPE__` Defined in GNU mode as the underlying type for `char32_t` (GNU version 40400 and above).

1.13.1 Pragmas

`#pragma` directives are used within the source program to request certain kinds of special processing. The `#pragma` directive is part of the standard C and C++ languages, but the meaning of any pragma is implementation-defined.

1.13.2 Instantiation Pragmas

The following are described in detail in the template instantiation section of this chapter:

```
#pragma instantiate
#pragma do_not_instantiate
#pragma can_instantiate
```

1.13.3 Precompiled Header Pragmas

The following are described in the section on precompiled header processing:

```
#pragma hdrstop
#pragma no_pch
```

1.13.4 Once Pragma

The front end also recognizes `#pragma once`, which, 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 front end 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 front end 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.

1.13.5 Diagnostic Pragmas

The following pragmas may be used to control the error severity used when a given diagnostic message is issued.

```
#pragma diag_suppress
#pragma diag_remark
#pragma diag_warning
#pragma diag_error
#pragma diag_default
#pragma diag_once
```

1.13 Predefined Macros

These are similar in function to the equivalent command-line options. Uses of these pragmas have the following form:

```
#pragma diag_xxx [=] error_number_or_tag, error_number_or_tag ...
```

The diagnostic affected is specified using either an error number or an error tag name. The “=” is optional. Any diagnostic may be overridden to be an error, but only diagnostics with a severity of discretionary error or below may have their severity reduced to a warning or below, or be suppressed. The `diag_default` pragma is used to return the severity of a diagnostic to the one that was in effect before any pragmas were issued (i.e., the normal severity of the message as modified by any command-line options). The following example suppresses the “pointless friend declaration” warning on the declaration of class A:

```
#pragma diag_suppress 522
class A { friend class A; };
#pragma diag_default 522
class B { friend class B; };
```

The `diag_once` pragma causes the specified diagnostics to be issued only once as a warning or remark.

1.13.6 Pack Pragma

When `USER_CONTROL_OF_STRUCT_PACKING` is `TRUE`, `#pragma pack` is recognized. It is used to specify the maximum alignment allowed for nonstatic data members of structs and classes, even if that alignment is less than the alignment dictated by the member’s type.

The basic syntax is:

```
#pragma pack(n)
#pragma pack()
```

where argument *n*, a power-of-2 value within the range defined by `TARG_MINIMUM_PACK_ALIGNMENT` and `TARG_MAXIMUM_PACK_ALIGNMENT`, is the new packing alignment that is to go into effect for subsequent declarations, until another `#pragma pack` is seen. The second form cancels the effect of a preceding `#pragma pack(n)` and either restores the default packing alignment specified by the `--pack_alignment` command-line option or, if the option was not used, disables the packing of structs and classes.

An enhanced syntax (intended to be compatible with the similar facility offered by Microsoft C/C++ compilers) is also supported in which keywords `push` and `pop` can be used to manage a stack of packing alignment values — for instance:

```
#pragma pack (push, xxx, 16)
/* Packing alignment set to 16. */
#include "xxx.h"
#pragma pack (pop, xxx)
/* Packing alignment reset to what it was prior to the
above #pragma pack directive, even if the included header
left some unrelated pragma pack entries on the stack. */
```

which has the effect saving the current packing alignment value and establishing a new one, processing the include file (which may leave the packing alignment with an unknown setting), and restoring the original value. Finally, the form

```
#pragma pack(show)
```

causes the front end to emit a warning describing the current setting of the maximum member alignment (if any).

In C++ a `#pragma pack` directive has “local” effect when it appears inside a function template, an instantiation of a class template, or the body of a member function or friend function that is defined within a class definition. When

such a context is entered, the current pack alignment state is saved, so that it can be restored when the context is exited. For example,

```
template <class T> class A {
:
#pragma pack(1)
:
};
#pragma pack(4)
main() {
    A<int> a;                // #pragma pack(1) is local to A<int>
    struct S { char c; int i; }; // sizeof(S) == 8, not 5
}
```

Moreover, within an instantiation of a class or function template, the default pack alignment is that which is in effect at the point of the template's definition, not at the point of its instantiation. Here is an example:

```
#pragma pack(1)
template <class T> int f(T) {
    struct S { char c; int i; }; // default pack alignment is 1 for each
    return sizeof(S);           // instance of f<T>
}
#pragma pack(4)
main() {
    int i = f(0);                // i = 5, not 8
}
```

These rules apply to inline-defined member functions, too, because their bodies are also scanned “out of order” relative to the textual order of the source program (i.e., not till all the class members have been declared).

```
#pragma pack(4)
struct A {
    int f() {
#pragma pack(1)                // #pragma pack(1) is local to A::f
        struct S { char c; int i; }; // sizeof(S) is 5
    }
    int g() {
        struct S { char c; int i; }; // sizeof(S) is 8
    }
#pragma pack(1)
    int h() {
        struct S { char c; int i; }; // sizeof(S) is 5
    }
};
```

When `IDENT_DIRECTIVE_AND_PRAGMA` is `TRUE`, `#pragma ident` is recognized, as is `#ident`:

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

Both are implemented by recording the string in a pragma entry and passing it to the back end.

When `PRAGMA_WEAK_ALLOWED` is `TRUE`, `#pragma weak` is recognized. Its form is

```
#pragma weak name1 [= name2]
```

where *name1* is the name to be given “weak binding” and is a synonym for *name2* if the latter is specified. The entire argument string is recorded in the pragma entry and passed to the back end.

1.14 Precompiled Headers

1.13.7 Microsoft Pragmas

The following pragmas are accepted in Microsoft mode:

```
#pragma push_macro("identifier")
#pragma pop_macro("identifier")
#pragma include_alias("a_long_file_name.h", "short.h")
```

In addition, when `NATIVE_MULTIBYTE_CHARS_SUPPORTED_WITH_UNICODE` is `TRUE`, the `setlocale` pragma is accepted:

```
#pragma setlocale("locale name")
```

The `pack` pragma is also accepted by the Microsoft compiler, but its acceptance by the front end is controlled by a separate configuration flag, not by whether or not Microsoft mode is being used.

1.14 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 EDG front end 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.

1.14.1 Automatic Precompiled Header Processing

When `--pch` appears on the command line, automatic precompiled header processing is enabled. This means the front end 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, here’s 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. This is configurable — see `PCH_DECL_SEQ_THRESHOLD`.

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 (i.e., when `USE_MMAP_FOR_MEMORY_REGIONS` is `FALSE`), 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 `HOST_ALLOCATION_INCREMENT`, 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.

1.14 Precompiled Headers

- 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.C
#include "xxx.h"
...           // Start of code
// b.C
#include "xxx.h"
...           // Start of code
```

When `a.C` is compiled with `--pch`, a precompiled header file named `a.pch` is created. Then, when `b.C` is compiled (or when `a.C` 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 front end 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 (see `PCH_FILE_SUFFIX`, which is set to `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.C": 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 front end 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 front end 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.C`) 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.

1.14.2 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.

1.14.3 Other Ways for Users 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 the user 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 (i.e., when `USE_MMAP_FOR_MEMORY_REGIONS` is `FALSE`) 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; and in addition:

- Command-line option `--pch_mem size` may be used to change the size of the preallocated block of memory from its default value. The argument *size* is the number of 1024-byte units. (`--pch_mem` must also be among the first options on the command line.)

1.14 Precompiled Headers

1.14.4 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. (An implementation can adjust `PCH_DECL_SEQ_THRESHOLD` to avoid creating precompiled header files with too little information in them.)

In general, it doesn't 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 doesn't 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.

The EDG front end source provides an example of how this can be done. A common idiom is this:

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

where `fe_common.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 `fe_common.h` is a bit over a megabyte in size. Another idiom, used by the source files involved in declaration processing, is this:

```
#include "fe_common.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 fifty-odd source files of the EDG front end share just six precompiled header files. If disk space were at a premium, one could decide to make `fe_common.h` pull in *all* the header files used — then, a single PCH file could be used in building the EDG front end.

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.

