1 Preface

The C/C++ compiler described in this manual is a native compiler that supports the C and C++ programming languages. This compiler can be called and controlled by means of the following commands:

`cc`, `c89` The compiler works as a C compiler.

`CC` The compiler works as a C++ compiler.

The C/C++ compiler is a part of the C/C++ development system CDS++, which is an independent software package from the Reliant UNIX base system. CDS++ includes not only the C/C++ compiler, but also the symbolic debugging aid DBX and various C++ libraries (i.e. the Cfront-compatible library, the standard C++ library and Tools.h++).

In order to use CDS++, you will also need the C libraries, headers and tools from the `ccs`, `attlib`, `atthead`, `ucblib`, `ucbhead`, `ucbcmd` and `attcmd` packages. These packages are part of the Reliant UNIX base system.

Depending on which license is purchased, the C/C++ compiler can be used as only a C compiler (with the “CDS” license) or as both a C and C++ compiler (with the “CDS++” license). See the Release Notice for further details.

1.1 Documentation for CDS++ and additional references

The following important manuals and reference literature are available for developing C and C++ programs using CDS++ and other software components of the Reliant UNIX system:

Manuals that can be ordered from SNI

– This “C/C++ Compiler” User Guide
  See the “Summary of contents” in the next section.

– “Reliant UNIX, Programmer´s Reference Manual”
  C library functions, system calls and commands for program development (e.g. `ld`, `sccs`, `yacc`)

– “DBX V2.3”, User Guide
  Symbolic Debugger for C, C++ and FORTRAN

– “Standard C++ Library V1.2”, User´s Guide and Reference
  Rogue Wave manual for the ANSI C++ libraries Strings, Containers, Iterators, Algorithms and Numerics

– “Tools.h++ V7.0”, User´s Guide and
  “Tools.h++ V7.0”, Class Reference
  Rogue Wave manuals for the widespread and commonly usable C++ class libraries

– “Cfront C++ Library”, Reference Manual
  Classes, functions and operators for complex math and stream-oriented I/O (compatible to Cfront V3.0.3)

Documentation supplied with CDS++

– “C++ Language Addendum” (see the Release Notice for details on the delivery format). This document contains ANSI C++ language elements which have not been defined in the C++ language definition by Bjarne Stroustrup (2nd. edition) or which deviate from it.

– Release Notice.
  The Release Notice (`readme` file) is a part of the CDS++ Readme package. It contains important information and the latest amendments related to the product, its use (especially on older SINIX V5.42 systems), and the supplied manuals.

Other reference literature and standards

- “The C Programing Language 2nd Edition - ANSI-C”, by B.W. Kernighan and D.M. Ritchie, Prentice Hall, ISBN 0-13-110370-9
  Introduction to C, problem-oriented description of the C language elements with a large number of examples, reference section (C language description), C solution.
- “The C++ Programming Language, 2nd Edition”, by Bjarne Stroustrup, Addison-Wesley, 1991, ISBN 0-201-53992-6
  Introduction to C and C++ with a large number of examples, three chapters on software development using C++ and a complete reference manual.
- “American National Standard for Information Systems - Programming Language C”, Doc.No. X3J11/90-013, February 14, 1990 or
  “International Standard ISO/IEC 9899 : 1990, Programming languages - C”
- “International Standard ISO/IEC 9899 : 1990, Programming languages - C / Amendment 1 : 1994”
- “System Application Binary Interface, MIPS Processor supplement”, Prentice Hall, ISBN 0-13-880170-3
- “Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++”,
  Doc.No. X3J16/96-0219R1, WG21/N0137, Dec 2 1996

  This document can be ordered from:
  American National Standards Institute (ANSI), Standards Secretariat: ITIC,
  1250 Eye Street NW, Suite 200, Washington DC 20005 USA)
  or from:
  Normenausschuß Informationstechnik im DIN
  Deutsches Institut für Normung e.V.
  10772 Berlin

1.2 Summary of contents

This User Guide is indented for C and C++ programmers. It describes how C and C++ programs can be compiled with the C/C++ compiler and then be linked and executed.

In order to work effectively with this manual, you will need to be familiar with the C and C++ programming languages and the Reliant UNIX operating system.

Following this preface, you will find an overview of the C/C++ Development System and its components described in chapter 2.

The commands to call the compiler are `cc`, `c89` and `CC`. These commands are described in detail with their possible options and effects in chapter 3.

Chapter 4 describes how headers can be precompiled.

Chapter 5 describes how programs can be optimized.

Chapter 6 describes how the link editor works and how you can create and use libraries.

Chapters 7 and 8 deal with extensions to the C and C++ language definitions and describe the implementation and machine-dependent language features of this compiler.

Chapter 9 (the appendix) contains an alphabetically-ordered list of all compiler options with references to the pages on which they are described.

1.3 Notational conventions

The following notational conventions are used to depict commands, options and program statements described in this User Guide:

`-B dynamic`	Uppercase letters, digits and special characters in `typewritten text` are constants and must be entered exactly as shown, except for the arguments for the `-K` option, which are shown in the manual in lowercase letters, but may be entered in both uppercase and lowercase (see page 23).
name	Lowercase letters in *italics* denote variables, which must be replaced by current values at the time of input.
{`cc` | `c89`}	Braces enclose alternatives from which one must be selected. The separator character | must not be specified.
[ ]	Square brackets enclose options that may be omitted.
( )	Parentheses are constants and must be specified.
␣	This symbol indicates that at least one white space character is necessary for the syntax.
...	Ellipses signify repetition, i.e. the preceding unit can be repeated several times in succession.

2 The C/C++ development system

The C/C++ development system CDS++ essentially consists of the following components:

- the C/C++ compiler
- extensive C++ libraries and runtime systems
- the symbolic debugger DBX

This chapter provides an overview of these components and their most important characteristics.

Detailed information on the C/C++ compiler can be found in the remaining chapters of this manual. For information on the various C++ libraries and the symbolic debugger DBX, see the manuals listed under the **Documentation** section.

2.1 The C/C++ compiler

2.1.1 General characteristics

Supported C and C++ language standards

The compiler supports the following C and C++ language standards:

- ANSI/ISO C with the ISO C Addendum 1 (1994)
- the status of the ANSI C++ draft from early 1996 including, in particular, exception handling, templates, new-style casts, name spaces, runtime type information (RTTI), etc.

Controlling the compiler

Three commands are available to call and control the compiler: `cc`, `c89` and `CC`.
The call syntax for these commands is based on the definition in the XPG4 standard.

`cc`, `c89` The compiler operates as a C compiler and optionally supports the Kernighan & Ritchie C, extended ANSI/ISO C (default for the `cc` command) and strict ANSI/ISO C (default for the `c89` command) modes.

`CC` The compiler operates as a C++ compiler and optionally supports the Cfront V3.0.3-compatible C++, extended ANSI/ISO C++ (default) and strict ANSI/ISO C++ modes.

The `cc`, `c89` and `CC` commands to call the compiler also include an integrated linkage phase with which compiled objects can be linked into an executable file or a shared library.

Compatibility

The following language modes are offered by the compiler to enable the migration or porting of older C and C++ applications: Kernighan&Ritchie C, Cfront C++ V3.0.3, preprocessor dialects based on Reiser’s cpp and Johnson’s pcc.

Portable software

To develop portable software, the compiler also supports the “strict” ANSI C and ANSI C++ modes. All deviations from the corresponding language standards are diagnosed in these modes.

64-bit data model

A new 64-bit data model that supports pointers and integers with a length of 64 bits (see Release Notice) is now available in addition to the 32-bit API.

High optimization

In order to produce applications with maximum runtime performance, a number of effective optimization techniques ranging from simple optimization to complex cross-module feedback optimization have been implemented in the compiler.

Native C++ compiler

In contrast to C++ V3.0, which generated C code from C++ code and passed it to the C compiler, CDS++ directly generates object code in ELF format from C++ code.

Precompilation of headers

In order to reduce the compilation time when compiling a source file, the compiler can store the compiled code of header files in a separate file and then reuse the precompiled header on compiling the same or any other source file.

Thread support

The compiler and the C++ libraries support threaded programs with the Thread package of the product DCE (Reliant UNIX) V2.0.

makefile dependencies

The compiler optionally generates file dependencies (with the `-M` option) that can be used as input for the UNIX program `make`.

Compilation listings

Source/error listings and cross-reference listings can be generated by the compiler on request and stored in corresponding listing files.

2.1.2 Structure of the compiler

The compilation of a source program is performed by the compiler in a number of sequential phases, where each phase is handled by a specific component of the compiler. Generally speaking, the compiler could be viewed as a combination of a compiler frontend, a compiler backend and a linkage phase. The frontend and backend are, in turn, made up of numerous subcomponents that are responsible for completing specific tasks. The most important compiler components are explained in brief below.

Compiler frontend

- Scanner

  The scanner performs the lexical analysis of the program, i.e. it reduces the program to the smallest units of the language (tokens) and checks whether these units correspond to the language scope of the compiler.

- Preprocessor

  The preprocessor resolves preprocessor statements in the source program and consequently performs the following tasks:

  – replacing names of constants and macros with corresponding values and texts (`#define` statements)

  – reading in header elements (`#include` statements)

  – preparing the conditional compilation of source program fragments (e.g. `#ifdef`/`#ifndef` statements)

  – processing of compiler-specific statements (`#pragma` directives)

  If the compiler is called with the `-P` or `-E` options, compilation is terminated after the preprocessor run (see page 31).

- Parser

  The parser performs a syntactic and semantic check of the source program based on results obtained from the scanning and preprocessor phases.

- Intermediate code generator

  If the C or C++ source text has no syntactic or semantic errors, it is compiled into intermediate code, which is then passed on to the compiler backend. If the optimization option `-F U` was specified, the intermediate code is optimized.

- Listing generators

  The listing generators generate the messages and listings requested by the user. See the section “Options to output listings and CIF information” on page 73.

Compiler backend

- "Ucode" generator and optimizer

  The intermediate source code generated by the compiler frontend is used to create additional intermediate code (Ucode) that is suitable for optimization across modules. If one of the optimization options `-O`, `-F O`*x* (where *x* = 2, 3, 4 or 5) is specified, this code is also optimized, i.e. the optimizer introduces changes in the object code in order to reduce the runtime of the executable program.

  If the `-F O`*x* options are specified in combination with `-c`, the unoptimized Ucode is placed in a file with the suffix `.o` (Ucode file).

  For further details on generating Ucode and on optimization, see the section "Optimization options" on page 51 and chapter 5 "Optimization" on page 91, respectively.

- Code generator

  The code generator creates Assembler source code from the intermediate source code.

  If the `-S` option is specified, the compiler run is terminated after the code generator has created the Assembler source code for each compiled source file and placed it in a file with the suffix `.s` (see page 31 for details).

- Assembler

  The Assembler translates the Assembler source code into the machine instructions of the processor on which your program is run. The generated machine instructions are stored in object files. For each source file named *file*`.c`, a separate object file named *file*`.o` is created on successful compilation.

  Object files generally consist of two segments, i.e. the text segment and the data segment. The text segment essentially contains program statements and can have read and execute privileges, but no write privileges. The data segment contains program data and is usually associated with read, write and execute privileges.

  If the `-c -K dual` options are specified, the generated object file *file*`.o` will contain the object code generated by the Assembler as well as the Ucode that is produced by the Ucode generator (dual object file).

Link editor

The link editor links the generated object files with one another and with the library functions that you call in your program. The linkage method depends on the selected linkage mode:

– Static linking:

  If you specify static linking in the `cc/c89/CC` call, a copy of the object file containing each of the library functions that you use in your program will be inserted into the executable at **link time**.

– Dynamic linking:

  If you specify dynamic linking in the `cc /c89/CC` call, the entire contents of the dynamically-linked library, which is referred to as a “shared object” below, will be mapped to the virtual address space of your process at **runtime** as soon as any function from that library is called in your program. During program execution, the external references in the program are associated with their related definitions.

For further details on linking, see the section “Link editor options” on page 65 and chapter 6 “The link editor” on page 111.

Schematic diagram of compiler components (Figure 1)

The following diagram (Figure 1) illustrates which compiler components are used in simple C compilations (`cc/c89` command). The compiler component used for C++ compilations (`CC` command) are analogous, but the name of the preprocessor output file would then be *file*`.I`.

The diagram does not show the various compiler components that are also involved when, for example, optimization based on “Ucode” occurs (see page 91ff) or when header files are precompiled and reused (see page 83ff). The additional output files and objects that can be generated by the compiler in such cases can be found under the descriptions of the respective options, especially in the sections “Options to select compilation phases” on page 30ff, “Link editor options” on page 65ff, and “Options to output listings and CIF information” on page 73, respectively.

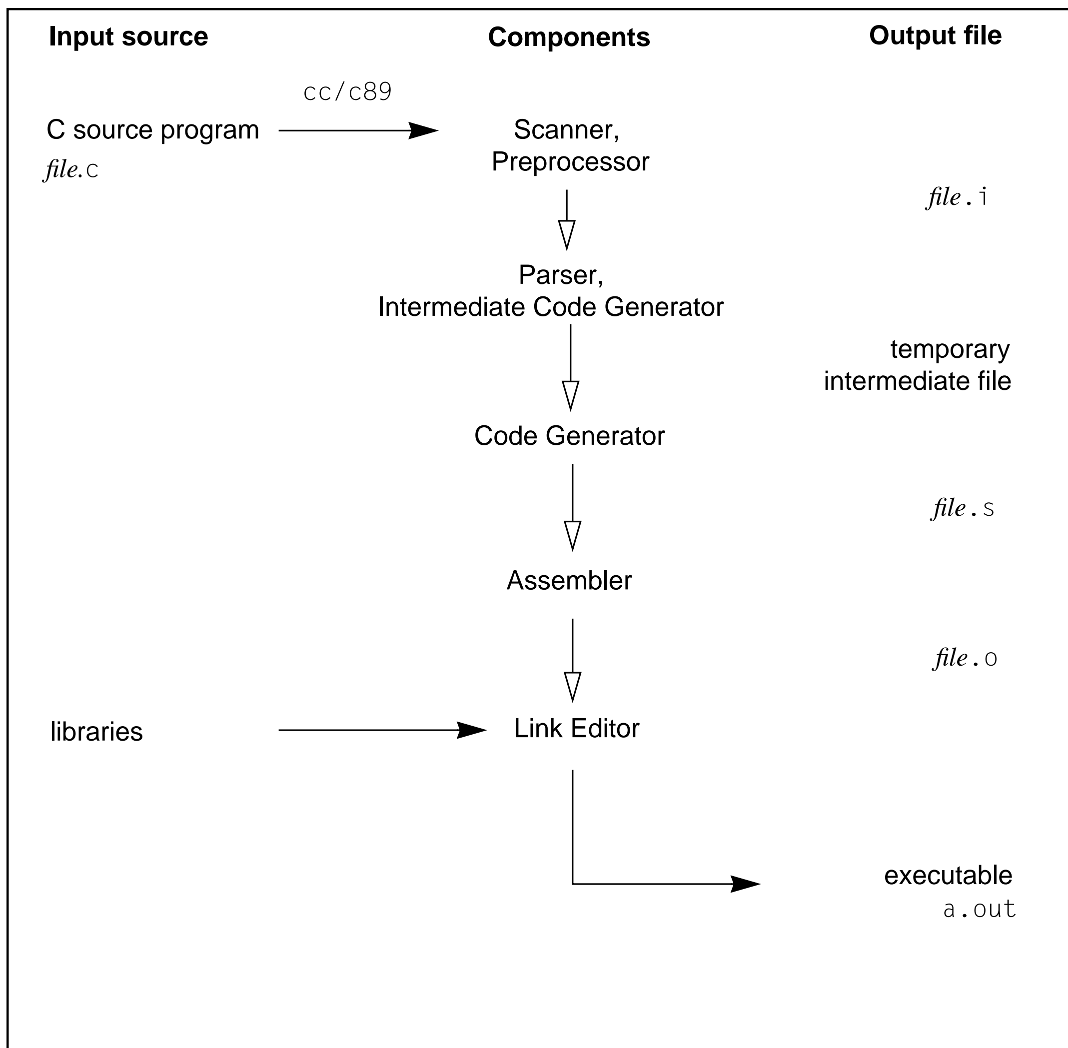


Figure 1: Compiler components used during compilation

2.2 The C++ libraries

The following C++ libraries are provided with CDS++:

– a standard C++ library based on the ANSI C++ draft

– a C++ library compatible with Cfront V3.0.3

– Tools.h++ V7.0

These libraries are provided in a statically linkable format and as shared objects. There is also a thread-safe version for each of the above libraries.

2.2.1 The standard C++ library

The standard C++ library can be used both in the ANSI C++ modes `-X w` and `-X e` and in the Cfront C++ mode `-X d` (to some extend). Depending on which C++ language mode is selected, this library is available in the following two variants:

Standard C++ library in ANSI C++ modes

In the ANSI C++ modes, the standard C++ library includes the following interfaces:

- A string class

  `<string>`

- Container classes

  `<bitset>`
  `<deque>`
  `<list>`
  `<map>`
  `<queue>`
  `<set>`
  `<stack>`
  `<vector>`

- Iterators

  `<iterator>`

- Generic algorithms

  `<algorithm>`

- Numeric classes and operations

  `<complex>`
  `<numeric>`

- I/O classes

  `<iostream.h>`
  `<fstream.h>`
  `<strstream.h>`
  `<stdiostream.h>`
  `<iomanip.h>`

  The I/O classes are currently not ANSI-compliant and correspond to the Cfront V3.0.3-compatible I/O library iostream.

The modules for the above interfaces are contained in the library `libCstd.a` or `libCstd.so`.

Additional interfaces that are required internally to implement C++ exception handling, for example, are available in a separate C++ runtime library.

All names of the standard C++ library, except for the I/O classes, are available in the `std` namespace.

The header files listed below, in which all ANSI C library functions are defined in the `std` namespace, constitute a further component of the standard C++ library. The names of these header files are derived from the names of the ANSI C headers as follows: each name is preceded by the letter `c`, and the `.h` suffix is dropped.

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<clocale>	<cstdarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<cstddef>		

No special specifications are required in the `CC` command to include these header files and to link the related modules.

Standard C++ library in the Cfront C++ mode

All ANSI C++ interfaces of the standard C++ library, except for the `<complex>` class (see also the ANSI C++ modes on page 14), can be used in the Cfront C++ mode.

The classes and functions for complex math and I/O are available in separate libraries (see the C++ Cfront library on page 17).

Since `namespaces` are not supported in the Cfront C++ mode, all names are in the global namespace and not in the `namespace std`.

The modules for the ANSI C++ interfaces are in the library `libCFstd.a` or `libCFstd.so`. This library must be explicitly specified with the following command when linking the program:

```
CC ... -X d -l CFstd
```

Otherwise, no special specifications are required in the `CC` command in order to include the headers and link the related modules.

Documentation

The standard C++ library is described in detail in the following manual:
“Standard C++ Library V1.2, User´s Guide and Reference” (one volume).

A description of the I/O classes compatible with Cfront V3.0.3 can be found in the “Cfront C++ Library”, Reference Manual.

2.2.2 The Cfront C++ library

The C++ library compatible with Cfront V3.0.3 can be used in the Cfront C++ mode `-X d`. This library includes the following interfaces:

- A class for complex math (`libcomplex.a` or `libcomplex.so`)

  `<complex.h>`

- Classes for stream-oriented I/O (`libC.a` or `libC.so`)

  ```
  <iostream.h>
  <fstream.h>
  <strstream.h>
  <stdiostream.h>
  <iomanip.h>
  ```

If complex math is used, the option `-l complex` must be specified at linkage. Otherwise, no special specifications are required in the `CC` command to include the headers and to link the related modules.

Since the standard C++ library does not currently contain any ANSI compliant I/O interfaces, the Cfront-compatible I/O classes are also used in the ANSI C++ modes. The modules needed for this purpose are integrated in the library `libCstd.a` or `libCstd.so` (see page 14).

Documentation

The Cfront C++ library is described in detail in the following manual:
“Cfront C++ Library”, Reference Manual

If you are using the earlier C++ compiler C++ V3.1B or V3.1C, you will also find a description of the library in the “C++ V3.1B, User Guide and Reference Manual”.

2.2.3 The Tools.h++ library

The Tools.h++ V7.0 library can be used in all C++ modes.

This library offers a broad spectrum of “foundation classes”, i.e.:

- string classes with pattern-matching mechanisms
- classes to handle the date and time
- virtual streams
- file and file manager classes
- container classes (collectable) with the option of implementing persistence and associated iterator classes:
  - Smalltalk-like container classes (without template usage)
  - template container classes to store values (`RWTVal`...)
  - template container classes to store pointers (`RWTPtr`...)
- classes for internationalization

In order to use this library, you will need to specify the option `-l RWtools` when linking the program in the ANSI C++ modes (`-X w`, `-X e`) or the option `-l CFtools` in the Cfront C++ mode.

Documentation

The Tools.h++ library is describe in detail in the following manuals:
“Tools.h++ V7.0, User´s Guide” and
“Tools.h++ V7.0, Class Reference”

2.3 The symbolic debugger DBX

DBX is an interactive symbolic debugger for programs written in the programming languages C and C++.

DBX is a **debugger** that enables software developers to search for errors in an executing program. It allows programmers to trace the execution of the program and to intervene if necessary.

DBX is a **symbolic** debugger, since software developers can work with source-code symbols when testing the program to be debugged, i.e. are not restricted to machine-code level.

DBX is an **interactive** debugger, since software developers can work interactively with it. DBX offers a character-based user interface and a graphical, OSF/Motif-based user interface.

DBX can also communicate with the SoftBench development environment by means of its graphical user interface.

When testing with DBX, you can:

- execute C and C++ programs under control,
- test executable files symbolically or at machine level,
- test very large applications by dynamically loading the symbol information,
- stop the test at predefined positions or under specified conditions,
- use a memory dump to check the position at which a program was aborted and to find this position in the source code,
- log the debugging run and the DBX session,
- log changes to variables,
- output symbol values and the contents of addresses in decimal, hexadecimal or octal format,
- change symbol values for test purposes,
- monitor parent and child processes simultaneously,
- test threaded programs,
- test exception handling in C++ programs and in
- SORBET applications.

Documentation

DBX is described in detail in the following manual:
“DBX V2.3 , Symbolic Debugger for C, C++ and FORTRAN, User Guide”

3 The cc, c89 and CC commands

3.1 Command syntax and general rules

{cc | c89 | CC} [*option*] ... *operand* ...

If you are invoking the c89 command, you must specify all options (see page 22) before entering operands (see page 23).
This "options before operands" sequence is not mandatory for the cc and CC commands. Other differences between the cc, c89, and CC commands are summarized below.

The cc, c89 and CC commands

cc

If the compiler is called with cc, it works as a C compiler, and the default language mode is set to extended ANSI C (see option -X a on page 34).
Options and operands may be specified in any order on the command line.
In contrast to the c89 command, -L *dir* is interpreted as an operand (see -L on page 24 and option -- on page 29).

c89

If the compiler is called with c89, it works as a C compiler, and the default language mode is set to extended ANSI C (see option -X c on page 34).
In this case, options and operands cannot be mixed on the command line, i.e. the "options before operands" sequence must be maintained.
In contrast to the cc/CC commands, -L *dir* is interpreted here as an option (see -L on page 24 and option -- on page 29).

CC

If the compiler is called with CC, it works as a C++ compiler, and the default language mode is set to extended ANSI C++ (see option -X w on page 35).
Options and operands may be specified on the command line in any order.
In contrast to the c89 command, -L *dir* is interpreted as an operand (see -L on page 24 and option -- on page 29).

Options

No *option* specified

If the source code contains no syntax errors, and all open references are resolved, the compiler generates an executable file `a.out`, which contains the executable program. The object code for each compiled source file is placed in a corresponding object file named *sourcefile*`.o`.

option

You can specify options in the compiler call to control the compilation process and to determine which arguments are passed to the programs for the individual compilation phases.

Options can also be used to instruct the compiler to perform only some of the compilation phases (see page 30ff). If the compilation process is not completed fully, all options that refer to the skipped compilation phases are ignored by the compiler. If multiple options are used to select the compilation phases to be performed, the compiler will stop after the earliest specified phase.

An option always consists of a single letter that is identified by a leading hyphen ("-").

Multiple options may be grouped, i.e. specified in succession after a single hyphen without any delimiting whitespace, only if none of the listed options take any arguments (e.g. `-V -c` could also be entered as `-Vc`).

Options that take arguments must be specified in accordance with the XPG4 Standard by separating the option and its argument with a space. This XPG4-compliant notation is strongly recommended, but is not enforced by the compiler for compatibility reasons (e.g. the compiler will accept `-ohello` instead of `-o hello`).

Arguments that contain delimiters (`:` or `,`) or the equals sign (=) must not be specified with any whitespace before or after these characters.

Examples

```
-D MACRO = 1      illegal
-D MACRO=1        legal
-R limit,  20     illegal
-R limit,20       legal
```

If the same option is specified more than once with conflicting arguments (e.g. `-K dual` and `-K no_dual`), the last option specified on the command line applies.

Options that are not known to the compiler, i.e. options that begin with an unrecognized letter after the leading hyphen ("-"), are passed through to the link editor `ld`. If the unknown option and the argument are separated by whitespace, the option is interpreted and passed as an option without an argument. In general, the option `-W l` (see page 69) should therefore be used whenever any options that are not known to the compiler are to be passed to the link editor `ld`.

Options with unrecognized arguments are ignored, and a corresponding warning is issued.

Special input rules for the `-K` *option*

`-K` *arg1*[,*arg2*...]
The `-K` option can be used to specify one or more arguments in succession, with a delimiting comma between each such argument. The delimiter between the arguments (i.e. the comma) must not be preceded or followed by any whitespace. Multiple `-K` options with one argument each have the same effect as a single `-K` option with multiple arguments delimited by commas. The arguments specified with the `-K` option may be entered in uppercase and/or lowercase letters (e.g. the arguments `PIC`, `pic`, `Pic`, etc. are equivalent). In the case of conflicting specifications (e.g. `-K uchar` and `-K schar`), the last entry is taken without issuing a warning.

Operands

The "operands" category includes the following entries:

- the names of input files, i.e.: *file.suffix*
- the link editor options `-l` *archive_code*, `-B dynamic` and `-B static`
- only for the `cc`/`CC` commands: also the link editor option `-L` *dir*

The compiler processes all options first, and then the operands, in the order in which they are specified on the command line.
All arguments that follow the `--` option (which ends the input of options) on the command line are interpreted as operands, even if they begin with a "-" character (see `--` on page 29).

file.suffix
Name of an input file.

The compiler determines the contents of a file, and thus the compilation steps to be performed in each case, from the file name extension. The file name must therefore have a suffix (or extension) that matches its contents. The possible suffixes that can be used to identify source files will depend on the mode in which the compiler is invoked and whether the compiler was called with the `cc`/`c89` command (C mode) or with `CC` (C++ mode).

The following suffixes are interpreted in individual cases as listed below:

`c` C source code (`cc`, `c89`) or C++ source code (`CC`) before the preprocessor run

`i` C source code (`cc`, `c89`) or C++ source code (`CC`) after the preprocessor run

`C`, `cpp`, `CPP`, `cxx`, `CXX`, `cc`, `CC`, `c++`, `C++`
C++ source code before the preprocessor run (`CC`)

`I` C++ source code after the preprocessor run (`CC`)

`s` Assembler source code

`o` Object file with object code, Ucode (Ucode file) or both (dual object file)

`a` Static library with object modules

`so` Shared dynamic library with object modules

In addition to the suffixes above, other user-defined suffixes to be recognized by the individual compiler components may be specified by using the `-Y F` option (see page 28).

File names with no suffix or an unrecognized suffix are passed through to the link editor without issuing a warning.

At least one input file (*file*`.`*suffix*) or one library in the form `-l` *archive_code* is required for each compiler call.

If more than one input file is specified, these files need not be of the same type, i.e. source files, Assembler files, and object files may all be specified in the same compiler call. In the case of object files and libraries, the order and position in which they are entered on the command line are significant for linking.

`-L` *dir*

`-L` *dir* is interpreted as an operand only when calling the compiler with the `cc` and `CC` commands. *dir* can be used to specify an additional directory that is to be searched by the link editor for the libraries specified with the `-l` option (see page 69 for more details).

`-l` *archive_code*
`-B dynamic`
`-B static`

These operands instruct the link editor to search for libraries named `lib`*archive_code*`.a` or `lib`*archive_code*`.so`, as determined by the `-B dynamic` and `-B static` entries if specified (see "link editor options" on page 65ff for details).

Exit status

0 Normal termination of the compiler run; no errors, but possibly with notes and warnings

1 Normal termination of the compiler run; with errors

2 Abnormal termination of the compiler run; with the occurrence of a fatal error

3.2 Description of options

The following sections describe the possible options for the `cc`, `c89` and `CC` commands in groups, depending on the context in which they are used. The options are classified as follows:

– General options (page 27)
– Options to select compilation phases (page 30)
– Options to select the language mode (page 34)
– Preprocessor options (page 37)
– Common frontend options in C and C++ (page 41)
– C++-specific frontend options (page 44)
– Optimization options (page 51)
– Options for object generation (page 59)
– Link editor options (page 65)
– Options to control message output (page 71)
– Options to select listings and CIF information (page 73)

The general aspects to be observed when entering options are discussed in the section on "Command syntax and general rules" (page 21).

A list of all options in alphabetic order with references to the pages on which they appear can be found in the appendix (page 199).

3.2.1 General options

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments for general control of the compilation run:

`verbose`
`no_verbose`

Note that the `-K verbose` specification, which causes additional information on template instantiation to be written to the standard error stderr, is presently only meaningful with the `CC` command.
`-K no_verbose` is the default setting.

`-o` *output_destination*

If this option is omitted, the compiler generates output files with default names and places them in the current directory.
The `-o` option can be used to rename the various output files of a compiler run and/or have them written to a different directory.
output_destination can be any of the following: only the name of a directory, only a file name, or a file name including directory components. The specified directories must already exist.

output_destination = directory name *dir*

The output files are created with default names and placed in the specified directory *dir* as follows:

- When an executable file, a shared object (option `-G`) or a prelinked object file (option `-r`) is generated, the file is assigned the name *dir*/`a.out`.
- If the `-c` option is specified, the object file is named *dir*/*sourcefile*`.o`.
- If the `-E` option is specified, the preprocessor output is written to the file *dir*/*sourcefile*`.i` (`cc`/`c89` command) or *dir*/*sourcefile*`.I` (`CC` command) instead of the standard output stdout.
- If the `-M` option is specified, the preprocessor output (dependency lines for further processing with `make`) is written to the file *dir*/*sourcefile*`.mk` instead of the standard output stdout.
- If the `-P` option is specified, the preprocessor output is written to the file `dir/`*sourcefile*`.i` (`cc`/`c89` command) or *dir*/*sourcefile*`.I` (`CC` command).
- If the `-S` option is specified, the Assembler source code is written to the file *dir*/*sourcefile*`.s`.

With the exception of the output files generated by the link editor (the executable file, shared object or prelinked object file), independent output files are created for each compiled source file in cases where multiple source files are specified for compilation.

output_destination = a specified *file_name* or
output_destination = a specified directory and file name: *dir/file_name*

If an executable file is being generated or if the `-o` option is specified in combination with option `-G`, `-r`, `-c`, `-E`, `-M`, `-P` or `-S`, the compiler writes the result to a file named *file_name* and places it in the current directory or in the directory specified with *dir*. Apart from the output file generated by the link editor (i.e. the executable file, shared object or prelinked object file), a different file name may be specified for all other output files, but only if a single source file is listed for compilation in each compiler call.

`-V`

For each compiler component that is implicitly called during the execution of `cc/c89/CC`, a separate line is written to the standard error stderr with the version, and possibly a Copyright note, for that component.

`-Y F`,*file-type*,*user_suffix*

This option can be used to define user-specific suffixes in the form *user_suffix* for input files of type *file-type* in addition to the standard suffixes (see page 23) recognized by the compiler.

The following entries are possible for *file-type*:

`c++`	C++ source file
`c`	C source file
`ass`	Assembler source code file
`obj`	Object file with ELF object code
`lib`	Static library
`shlib`	Dynamic library

Example

```
-Y F,ass,asm
```

This entry instructs the compiler to accept any input file named *file*`.asm` as an Assembler source code file.

`--`

This option ends the input of options, i.e. causes all following arguments (except for the link editor options that fall under the "operands" category) to be interpreted as file names, even if they begin with a hyphen. This makes it possible to specify file names that start with a hyphen (e.g. `-hallo.c`).
The following link editor options are an exception and are permitted after the `--` option:
`-l` *archive_code*
`-B dynamic`
`-B static`
`-L` *dir* (only with the `cc` and `CC` commands; in the `c89` command, this entry would be interpreted as a file name!)

3.2.2 Options to select compilation phases

All of the options listed below always suppress the linkage run and cause any link editor options and operands that may have been specified to be ignored.

`-c`

This option terminates the compiler run after an object file named *file*`.o` has been generated for each compiled source file. The object file is written to the current directory by default. If desired, the `-o` option (see page 27) may be used to specify another file name and/or directory.
Depending on which additional option is also specified at the same time, the object file will contain object code (in ELF format), Ucode or both. You will find more general information on Ucode in the section "Ucode and dual object files" on page 97.

`-c` without and additional option
The .o file contains object code in ELF format and can be linked into an executable file by the link editor.

`-c -F O`*x* (*x* = 2, 3, 4 or 5)
The .o file contains only Ucode (and is hence called a "Ucode file"). Ucode files can be subsequently processed further, i.e. optimized and linked, only with the `cc/c89/CC` command and by repeating a `-F O`*x* optimization option (see page 51ff for details).

`-c -K dual`
The .o file contains both object code as well as Ucode (and is hence called a "dual object file"). The object code in a dual object file can be processed further (i.e. linked) like any conventional ELF object. The Ucode of a dual object file can be processed further with another `cc/c89/CC` command and the `-F O`*x* options (page 51ff). See also `-K dual` on page 59.

`-c -q f`
The .o file contains object code in ELF format with profiling information. This object file can be subsequently linked into a profiling executable by repeating the `cc/c89/CC` command with the `-q f` option. See also `-q f` on page 62.

`-c -K dual -q f`
The .o file contains both object code as well as Ucode.
The object code of this dual object file contains profiling information and can be subsequently linked into a profiling executable by repeating the `cc/c89/CC` command with the `-q f` option. See also `-q f` on page 62.
The Ucode can be processed further by repeating the `cc/c89/CC` command with the `-F O`*x* options (see page 51ff).

`-E`

The compiler run is terminated after the preprocessor phase, and the result is written to the standard output stdout. Any blank lines present in the file are combined in the process, and the corresponding `#line` directives are generated. By default, C and C++ comments are removed from the preprocessor output (see option `-C` on page 37). If the `-o` option is specified (see page 27), the result of the preprocessor run is written to a file instead of the standard output stdout.

`-M`

The compiler run is terminated after the preprocessor phase; however, instead of the normal preprocessor output (cf. `-E`, `-P`), a list of dependency lines that is suitable for further processing with the UNIX `make` program is generated and written to the standard output stdout. If the `-o` option is specified (see page 27), the file dependency list is written to a file instead of the standard output stdout.

`-P`

The compiler run is terminated after the preprocessor phase, and the result is written to a file named *file*`.i` (`cc`/`c89` command) or *file*`.I` (`CC` command) and placed in the current directory instead of the standard output stdout as in the `-E` option.
The output does not contain any additional `#line` directives. By default, C or C++ comments are removed from the preprocessor output (see option `-C` on page 37).
file`.i` can be subsequently compiled further with the `cc`/`c89`/`CC` commands, whereas *file*`.I` can only be compiled with the `CC` command. If desired, the `-o` option (see page 27) may be used to specify another file name and/or directory.

`-S`

The compiler run is terminated after generating Assembler source code for each compiled source file. The Assembler source code is created for each specified source file and placed in a file named *file*`.s` in the current directory. If desired, the `-o` option (see page 27) may be used to specify another file name and/or directory.
The C/C++ source code is copied into the Assembler source code as comments.
Please note that if the `-O` optimization option is specified at the same time, the inserted C/C++ source code cannot always be placed in the correct position. If cross-module optimizations (`-F O`*x*) are involved, the result cannot be stored as an Assembler source code file. The `-F O`*x* option is ignored if specified in combination with `-S`.
file`.s` can be subsequently compiled further with the `cc`/`c89`/`CC` commands.

`-y`

This option can only be specified with the `CC` command.
The compiler run is terminated after the prelinker phase (automatic template instantiation), and an object file named *sourcefile*`.o` containing the instantiated templates is generated for each compiled source file. This is useful for objects that are to be subsequently incorporated in a (`.a` or `.so`) library or in a prelinked object file. No automatic instantiation is performed for templates within libraries or prelinked object files. Note that the `-y` option can only be meaningfully used in the default automatic instantiation mode (`-T auto`).

Example

Contents of the source files (fragments):

```
// a.h:

class A {int i;};

// f.h:

template <class T> void f(T)

{
   /* Any code */
}

// b.c:

#include "a.h"
#include "f.h"

void foo() {
   A a;
   f(a);
}

// main.c:

void main(void)
{
   foo();
}
```

Commands:

```
CC -c b.c
```

The first compilation produces an object file `b.o` and a template information file `b.o.ii`, where each contains an entry that the function `f(A)` is not instantiated.

```
CC -y b.o
```

The `b.o` and `b.o.ii` files generated in the first compilation run are updated, and the function `f(A)` is instantiated.

```
ar -r x.a b.o
```

The module in `b.o` is added to the library `x.a`.

```
CC main.c x.a
```

An executable file named `a.out` is generated.

The following command sequence, by contrast, would not produce the desired result:

```
rm *.o *.ii *.a a.out   /* Cleanup the current directory */
CC -c b.c
ar -r x.a b.o
CC main.c x.a
```

This command sequence results in an error message. This is because the function `f(A)` cannot be found, since no automatic instantiation is performed for the templates in the library `x.a`.

3.2.3 Options to select the language mode

`-X t`
`-X a`
`-X c`

These options are used to select the C language mode and can only be specified when calling the compiler with `cc` and `c89`.

`-X t`

K&R C mode.
This mode should not be used for new developments. It is typically intended for porting "old" K&R C sources and/or systematic conversions to ANSI C.
The compiler accepts C code, as defined by Kernighan&Ritchie in the reference manual ("The C Programming Language", First Edition). It also supports C language elements of the ANSI C standard that are semantically identical to the Kernighan&Ritchie "definition" of the C language (e.g. function prototypes, `const`, `volatile`). This simplifies the conversion of a K&R C source to ANSI C. All C library functions of the system (i.e. ANSI functions, POSIX and X/OPEN functions, UNIX extensions) are available for use.
As far as the preprocessor behavior is concerned, ANSI/ISO C is the default. If desired, the option `-K kr_cpp` can be specified to convert the preprocessor behavior to K&R C (as required when porting old C sources, for example).
`__STDC__` and `__STDC_VERSION__` are set to the value 0.

`-X a`

Extended ANSI C mode (the default setting when calling the compiler with `cc`).
The compiler supports C code, as defined in the ANSI/ISO C standard, including the ISO C Amendment 1. In addition, various other language extensions are also supported (see chapter 7). Note that the name space is not restricted to names specified by the standard. All C library functions (ANSI functions, POSIX and X/OPEN functions, and UNIX extensions) may be used.
`__STDC__` and `__STDC_VERSION__` are set to the value 0.

`-X c`
Strict ANSI C mode (the default when calling the compiler with `c89`).
This mode can be used to test a program for ANSI/ISO conformance.
As in the extended ANSI C mode (`-X a`), the compiler supports C code in accordance with the ANSI/ISO C standard.
However, in contrast to the extended ANSI C mode, the name space is restricted to the names defined in the standard, and only the C library functions that are defined in the ANSI/ISO standard are available.
Deviations from the standard result in compiler messages (mostly warnings). If desired, the output of errors can be forced in such cases by specifying the option
`-R strict_errors`.
`__STDC__` has a value 1, and `__STDC_VERSION__` a value of 199409L.

`-X d`
`-X w`
`-X e`
These options are used to select the C++ language mode and can be specified when calling the compiler with `CC`.

`-X d`
Cfront C++ mode.
This mode is only offered for compatibility reasons and should not be used for new developments. It supports the C++ language elements of Cfront V3.0.3 (see "The Annotated C++ Reference Manual" by Margaret A. Ellis and Bjarne Stroustrup). Cfront V3.0.3 was last released with the compiler C++ V3.1B/C.
The following C++ libraries are available:
- the Cfront C++ library (complex math and stream-oriented I/O)
- a limited version of the standard C++ library (Strings, Containers, Iterators, Algorithms, Numerics)
- the Tools.h++ library

More information on the C++ libraries can be found in the section "The C++ libraries" on page 14.
C++ sources must be compiled and linked with `-X d` if their objects are to be linkable with C++ V3.1B/C objects.
`__STDC__` has a value of 0, and `__cplusplus` has a value of 1.

`-X w`

Extended ANSI C++ mode (the default when calling the compiler with `CC`).
The compiler supports C++ code in accordance with the definition proposed in the ANSI C++ draft for the future ANSI/ISO C++ standard (see the References in the appendix). In this case, the name space is not restricted to names specified in the standard.
The following C++ libraries are available:

- the standard C++ library (Strings, Containers, Iterators, Algorithms, and Numerics), including the Cfront-compatible I/O classes
- the Tools.h++ library

For more information on C++ libraries, see the section "The C++ libraries" on page 14.
As in the extended ANSI C mode (`-X a`), various language extensions (see chapter 7) as well as all C library functions of the system are available for use.
`__STDC__` has a value of 0, and `__cplusplus` has a value of 2.

`-X e`

Strict ANSI C++ mode.
In terms of the C++ language support (based on the ANSI/ISO C++) and the available C++ libraries, this mode corresponds to the extended ANSI C++ mode (`-X w`).
However, in contrast to the extended ANSI C mode, only the C library functions defined in the ANSI/ISO standard are available.
Deviations from the standard result in compiler messages (mostly warnings), but the output of errors can be forced in such cases by specifying the option
`-R strict_errors`.
`__STDC__` has a value of 1, and `__cplusplus` has a value of 199504L (which may change in future versions; see the Release Notice for details).

3.2.4 Preprocessor options

`-A` *name*(*tokens*)
: This option can be used to define an assertion, as if by a preprocessor `#assert` directive (see the section on "Preprocessor directives" on page 149).

`-A -`
: If this option is specified, all predefined preprocessor macros (except for those which begin with __) and all predefined preprocessor assertions are deleted (see the section on "Predefined preprocessor names" on page 78). This option does not affect macros and assertions that were defined by the user with options or preprocessor directives.

`-C`
: This option is evaluated only if the `-E` or `-P` option is also specified (see page 31). It causes C or C++ comments to be retained in the preprocessor output. Such comments are removed by default.

`-D` *name*[=*value*]
: This option can be used to define names, symbolic constants and macros (as if by a preprocessor `#define` directive).
`-D` *name* corresponds to the `#define` directive for defining names, i.e. `#define` *name*;
`-D` *name*=*value* corresponds to the `#define` directive for text substitutions,
i.e. `#define` *name value.*

`-H`
: Causes a list of all header files used during the compilation run to be written to the standard error stderr.

`-I` *dir*
: *dir* is added to the list of directories that are searched by the preprocessor for header files. If multiple directories are specified by entering this option more than once, they are searched for header files in the same order in which they are listed.

If the relative pathname of the header file (which does not begin with a slash /) is specified in the `#include` directive enclosed within quotes "...", the preprocessor searches the directories in the following order:

1. the directory of the source or header file containing the `#include` directive
2. the directories that were specified with the preprocessor option `-I`

3. either the directories specified with the option `-Y I` (see page 39) or the standard directories, as listed under a) to f) below.

   Standard directories, which are searched last:

   a) Only with the `CC` command in Cfront C++ mode (`-X d`): the directory `/opt/CDS++/include/CF`

   b) Only with the `CC` command in ANSI C++ modes (`-X w` and `-X e`): the directory `/opt/CDS++/include/CC`

   c) In all cases: the directory `/opt/CDS++/include/C`

   d) Only when the `-K thread` option is specified: the directory `/opt/thread/include`

   e) Only in C++ modes: the directories `/opt/CDS++/include/C/sysincl` and `/opt/CDS++/include/C/sysincl/SYS`
      (in the newer Reliant UNIX versions, these directories are actually links to `/usr/include` or `/usr/include/sys`)

   f) In all cases: the standard directories of the system `/usr/include` and `/usr/include/sys`

If the relative pathname of the header file is specified in the `#include` directive enclosed within angular brackets <...>, the preprocessor will only search the directories listed under points 2 and 3 above.

If you want the preprocessor to search some other directories last instead of the standard directories listed above, you can specify such directories by using the `-Y I` option (see page 39).

`-K` *arg1*[,*arg2*...]
General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control preprocessor behavior:

`ansi_cpp`
`kr_cpp`
`-K ansi_cpp` is the default setting in all C and C++ language modes of the compiler. This means that preprocessor behavior in accordance with the ANSI/ISO C standard is also supported in the K&R C mode.
The obsolete preprocessor behavior based on Reiser cpp and Johnson pcc can be turned on with `-K kr_cpp`, but only in K&R C mode. The `-K kr_cpp` specification is ignored in all other C and C++ language modes.

`-U` *name*

Undefines a macro or a symbolic constant *name* (as when using the preprocessor directive `#undef`), where *name* is a predefined preprocessor name (see page 78) or a name that was defined with the `-D` option in the command line before or after option `-U` . This option has no effect on `#define` directives in the source program.

`-Y I,`*dir*[:*dir*...]

Instructs the preprocessor which directory or directories are to be searched for header files last. *dir* specifies the directory.
Without this option, the last directories to be searched are the standard directories (see option `-I`, points a) to f)).

`-Z pch`
`-Z create_pch,`*file_name*
`-Z use_pch,`*file_name*

These options can be used to control whether the compiler is to create and use precompiled headers (PCH files) in accordance with its own algorithm ("automatic") or as specified by the user ("manually").
Note that only one source file per compiler call can be compiled when creating and using PCH files.
More detailed information and examples of PCH files can be found in chapter 4.

`-Z pch`
Automatic PCH mode.
The compiler creates and uses precompiled headers (PCH files) on the bases of the following principle:

– If there is no PCH file (file with the suffix `.pch`) or none that is suitable in the current directory, the compiler creates a PCH file named *sourcefile*`.pch`.
– If one or more PCH files are present in the current directory, the compiler uses the PCH file that is suitable for each case. Note that the basename *xxx* of the reused PCH file *xxx*`.pch` need not be identical to the name of the source file.

Example

The source file `A.c` is compiled with the option `-Z pch`, and the current directory contains the PCH files `A.pch`, `B.pch` and `C.pch`. When searching for a suitable PCH file, the compiler will use any PCH file that is suitable for the source file (e.g. `C.pch`) and not necessarily the one with the same basename as the source file (i.e. `A.pch`).

`-Z create_pch`,*file_name*
Manual PCH mode.
The compiler generates a PCH file with the specified name.

`-Z use_pch`,*file_name*
Manual PCH mode.
The compiler uses the PCH file with the specified name. If this PCH file is not suitable for the source file, the compiler issues a warning and recompiles the appropriate header files.

`-Z pch_dir`,*dir*
This option can be used in the automatic and manual PCH modes (see `-Z pch`, `-Z create_pch`, `-Z use_pch`) to define another directory for creating and reusing PCH files.

`-Z pch_messages`
`-Z no_pch_messages`
If `-Z pch_messages` is specified, the compiler issues a message to confirm that a PCH file has been created or used. For example:
`test.C : CFE1633 creating precompiled header file "test.pch"`
or
`test.C : CFE1632 using precompiled header file "test.pch"`

`-Z no_pch_messages` is the default setting.

3.2.5 Common frontend options in C and C++

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control the compiler frontend in the C and C++ modes:

`uchar`
`schar`

The data type `char` is `unsigned` by default. If `-K schar` is specified, `char` is treated as a `signed char` in expressions and conversions.
Note that the use of this option may result in portability problems!

`at`
`no_at`

`-K at` allows the use of the "at" sign '`@`' ' in identifiers.
`-K no_at` is the default setting for ANSI C/C++.

`dollar`
`no_dollar`

`-K dollar` allows the use of the "dollar" sign '`$`' in identifiers.
`-K no_dollar` is the default setting for ANSI C/C++.

`mb`
`no_mb`

The `-K mb` option turns on multibyte character recognition in the compiler. Since multibyte routines slow down the compilation process, this feature is turned off by default.

`signed_fields_signed`
`signed_fields_unsigned`

If `-K signed_fields_unsigned` is specified, `signed` bit fields are always interpreted as `unsigned`. This option is only offered for compatibility with older C-DS versions and is only meaningful in K&R C mode.
`-K signed_fields_signed` is the default setting.

`plain_fields_signed`
`plain_fields_unsigned`

These arguments control whether integer bit fields (`short`, `int`, `long`, `long long`) are treated as `signed` or `unsigned` types by default.
According to the ABI, `-K plain_fields_signed` is the default.

`long_preserving`
`unsigned_preserving`

These arguments control whether arithmetic operations with operands of type `long` and `unsigned int` return a result of type `long` (`long_preserving`) in accordance with K&R mode (first edition; see section 6.6 in the appendix) or of type `unsigned long` (`unsigned_preserving`) in accordance with ANSI/ISO C.
`-K unsigned_preserving` is the default setting.

`bit32`
`lp64`

These arguments are used to select the 32-bit or 64-bit data model. All modules that are linked into an executable must use the same data model. According to the ABI, the 32-bit data model is the default. The data types are aligned in accordance with their length (see below) on a byte, halfword, word or double-word boundary.
Depending on the data model, an assert predicate called `data_model(bit32)` or `data_model(lp64)` is predefined. For `-K lp64`, there is also a corresponding `__LP64__` macro.

Length of data types in bits:

Type	bit32	lp64
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
Pointer	32	64
ptrdiff_t	32	64
size_t	32	64
wchar_t	32	32

An object generated with `-K lp64` can only be executed on Reliant UNIX systems that support 64-bit data access. More information on this subject can be found in the Release Notice.

`alternative_tokens`
`no_alternative_tokens`

These arguments control whether alternative tokens are to be recognized by the compiler.
This includes digraph sequences (e.g. `<:` for `[`) in the C and C++ modes, and additional keywords for operators (e.g. `and` for `&&`, `bitand` for `&`) which are only valid in the C++ language mode.
`-K alternative_tokens` is the default for the ANSI C++ modes;
`-K no_alternative_tokens` is the default for all other modes.

`longlong`
`no_longlong`

These arguments control whether or not the data type `long long` is recognized by the compiler.
`-K longlong` is the default, i.e. the preprocessor define `_LONGLONG` is set in this case.
If `-K no_longlong` is specified, any use of the type `long long` will result in an error.

3.2.6 C++-specific frontend options

The options described in the following sections on "General C++ options" and "Template options" are only applicable to the `CC` command.

General C++ options

General C++ options can be used to control the following C++ features:

– whether tables are defined for virtual class functions
– whether the keywords `wchar_t` and `bool` are recognized
– the scope of initialization statements in `for` loops
– whether the old specialization syntax is accepted

Apart from the definition of tables for virtual functions, none of the language features listed above are supported in Cfront C++ mode.

`-K` *arg1*`[,`*arg2*`...]`

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control the C++ frontend:

`normal_vtbl`
`force_vtbl`
`suppress_vtbl`

These arguments allow you to specify how the virtual function table is to be defined by the compiler.

`-K normal_vtbl` (default)
The virtual function table of a class is defined in the compilation unit as soon as the compiler encounters the first virtual function of the class that is not declared `inline` or as a pure virtual function. In the case of classes that do not contain such virtual functions, the virtual function table is defined as a local static data area.

`-K suppress_vtbl`
No virtual function table is defined if the corresponding classes do not include any virtual functions.

`-K force_vtbl`
Virtual function tables are defined in any case, i.e. also for classes that do not contain any virtual function. In contrast to the default behavior (`-K normal_vtbl`), the tables are not defined as local static data areas.

`using_std`
`no_using_std`

These arguments determine the use of ANSI C++ library functions for which names have been defined in the standard name space `std`.

If `-K using_std` is specified, the compiler behaves as if the following lines were entered at the start of a compilation unit:

```
namespace std{}
using namespace std;
```

`-K using_std` is the default in extended ANSI C++ mode (`-X w`).

`-K no_using_std` is the default in strict ANSI C++ mode (`-X e`) and the only possible behavior in Cfront C++ mode (`-X d`).
If `-K no_using_std` is set in the extended or strict ANSI C++ mode, the source program must contain the statement `using namespace std;` otherwise, the names must be qualified appropriately before the first call to an ANSI C++ library function.

`wchar_t_keyword`
`no_wchar_t_keyword`

These arguments can be used to define whether `wchar_t` is recognized as a keyword.

`-K wchar_t_keyword` is the default in the ANSI C++ modes. In this case, the preprocessor macro `_WCHAR_T` is defined.

`-K no_wchar_t_keyword` is the default and the only possible behavior in the Cfront C++ mode.

`bool`
`no_bool`

These arguments can be used to define whether `bool` is recognized as a keyword.

`-K bool` is the default in the ANSI C++ modes. In this case, the preprocessor macro `_BOOL` is defined.

`-K no_bool` is the default (and only possible) behavior in the Cfront C++ mode.

`old_for_init`
`new_for_init`

These arguments define how an initialization statement in `for` and `while` loops is to be treated.

`-K old_for_init`
Specifies that an initialization statement has the same scope as the entire loop. This is the default setting in the Cfront C++ mode.

`-K new_for_init`
Specifies the new ANSI C++-conformant scope rule, which surrounds the entire loop in its own implicitly generated scope.
This is the default setting in the ANSI C++ modes.

`old_specialization`
`no_old_specialization`

These arguments are used to enable or disable acceptance of new template specializations `template<>` syntax.

`-K old_specialization` is the default setting in the Cfront C++ mode. In this case, the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 1.

`-K no_old_specialization` is the default setting in the ANSI C++ mode. In this case, the compiler implicitly defines the macro `__OLD_SPECIALIZATION_SYNTAX` with the value 0.

Template options

`-T none`
`-T auto`
`-T local`
`-T all`

These options control how templates with external linkage are instantiated. This includes function templates as well as (non-static and non-inline) functions and static variables that are members of template classes. These templates types are combined under the generic term "template entity" below.
More detailed information on the individual options can be found in the section “Overview of instantiation modes” on page 165.

`-T none`
Template entities are instantiated only in cases where an explicit instantiation request or an `instantiate` pragma exists.

`-T auto`
Template entities are instantiated in cases where an explicit instantiation request or an `instantiate` pragma exists, and further template entities to be instantiated are selected "automatically" by the compiler on the basis of one of its algorithms. A detailed description of the algorithm is presented under "Automatic instantiation" in chapter 8.

`-T auto` is the default setting.

`-T local`
All template entities used in a compilation unit are instantiated, with internal linkage for the functions generated in the process.

`-T all`
All template entities that are used or declared in a compilation unit are instantiated.

`-T max_iterations,`*n*
Specifies the maximum number (*n*) of prelinker iterations in automatic instantiation mode (`-T all`). The default is *n* = 30.

`-T add_prelink_files`,*pl_file1*[,*pl_file2*...]

This option can be used to specify objects and libraries that are taken into account as described below when the prelinker determines the instances to be generated:

pl_filei is the name of an object file (`.o` file), a static archive (`.a` file) or a shared library (`.so` file).

- If an object file or library *pl_filei* contains the definition of a function or static data member, no template instance that is a duplicate of that function or static data member is generated.
- If an object file or library *pl_filei* needs template instances, these instances are not generated.

Problems:

Example 1

The library `libX.so` contains the use of a function that is declared as a template. If the objects of this library are instantiated without the knowledge of other libraries, the prelinker will generate an instance for this template function. The library `libX.so` is, however, to be subsequently used together with the library `libY.so`, which contains a definition of this function, so the instance should not be generated in `libX.so`.

Example 2

The libraries `libX.a` and `libY.a` contain references to the same template instances. If the objects of these two libraries are both preinstantiated with the `-y` option, this will result in duplicates.

In such cases, the prelinker must be given a hint that symbols are to be defined elsewhere and that no instances should hence be generated. This can be done by using the option `-T add_prelink_files`.

Solutions:

Example 1

The objects of the library `libX.so` are preinstantiated with the `-y` option, but the option `-T add_prelink_files,libY.so` informs the prelinker that no duplicates for `libY.so` are to be generated.

Example 2

To begin with, the objects of the library `libX.a` are first preinstantiated with the `-y` option. This is followed by the preinstantiation of objects of the library `libY.a`, but the option `-T add_prelink_files,libX.a` is used here to inform the prelinker that the library `libX.a` needs to be considered and that no duplicates for `libX.a` should be generated.

`-T rem_prelink_libs,`*lib1*`[,`*lib2*...]

When generating an executable file, *libi* can be used to specify the name of a shared library that is **not** to be searched by the prelinker for existing definitions. If the library *libi* contains the definition for an instance, this instance will be generated by the prelinker if it is required in the program part to be linked and does not exist. The library *libi* will then only be passed to the linker.

This option is useful in automatic instantiation mode, since it enables shared libraries to be exchanged without always having to recompile the applications that use them.

Problem:

The first version of a shared library `libX.so` contains the definition of a template instance that is needed from the main program. Consequently, the instance will not be generated for the main program.
The next version of the shared library no longer contains the definition of the template instance. This results in unresolved external references when the main program is called without repeating the prelinker and linkage run.

Solution:

Such problems can be avoided by specifying the option `-T rem_prelink_libs` when linking the program:
`CC` *objects* `-l X -T rem_prelink_libs,libX.so`

This option could, however, result in duplicate definitions and should therefore be used with extreme caution.

`-K` *arg1*`[,`*arg2*`...]`

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control template instantiation:

`assign_local_only`
`no_assign_local_only`

These arguments determine whether or not the assignment of instantiations is only supported locally. If `-K assign_local_only` is set, the following applies:

- Instantiations can only be assigned to object files that are located in the current directory (local files).
- Instantiations can only be assigned to an object file if the current directory at the time of the instantiation matches the current directory at compile time.

Example

```
cd dir1          # The current directory when
CC -c test1.c    # compiling test1.c is dir1

cd ../dir2       # The current directory when
CC -c test2.c    # compiling test2.c is dir2

cd ../dir1       # The current directory for the
                 # prelinker is dir1
CC -K assign_local_only -o test test1.c ../dir2/test2.o
```

In this example, the assignment of instantiations is restricted to the local object file `test1.o`.

`-K no_assign_local_only` is the default setting.

`implicit_include`
`no_implicit_include`

These arguments determine whether the definition of a template is implicitly included (see "Implicit inclusion" on page 169)

`-K implicit_include` is the default setting.

`instantiation_flags`
`no_instantiation_flags`

The default setting, i.e. `-K instantiation_flags`, causes special symbols to be generated for use by the prelinker during automatic instantiation.
If `-K no_instantiation_flags` is set, no such symbols are generated, so the object size is reduced. Consequently, no automatic instantiation with `-T auto` is possible in this case.

3.2.7 Optimization options

The compiler offers a wide variety of optimization options, which include a core group of options for setting optimization levels (see `-O`, `-F O2`, `-F O3`, `-F O4`, `-F O5`, `-F X` and `-FX4` below). The remaining optimization options, some of which are highly specialized, are described together under the section "Other optimization options" (starting on page 55).

More detailed information and examples can be found in the chapter on "Optimization" (page 91).

Options to select optimization levels

`-O`

The optimizer (uopt) performs a number of global optimization measures, which apply to the individual compilation unit. These measures include:

- Elimination of common subexpressions
- Elimination of partial redundancies
- Propagation of constant expressions
- Branch optimization
- Loop optimization (e.g. removal of invariant portions in loops, conversion of time-consuming operations into more efficient ones, etc.)
- Dead code elimination
- Loop unrolling
- Register allocation per procedure

Loop unrolling can be controlled with the options `-F loopunroll,`*n* (see page 57) and `-F unrolllimit,`*n* (see page 58).
The `-O` option cannot be specified in combination with the debugging option `-g` (see page 59).

A higher level of optimization, which includes special measures that extend across modules, can be activated with the option `-F O`*x* (where *x* = 2, 3, 4 or 5).

`-F O2`

Enables all global optimization measures across modules (see option `-O`, page 51), i.e. across all compilation units, and also the inline expansion of user functions.

If this option is specified in combination with `-c`, the output files (.o files) will contain only Ucode. These "Ucode files" can be subsequently processed further (i.e. optimized and linked) only with the `cc`/`c89`/`CC` command and by repeating the `-F O2` option (see also the option `-F ucode`). Ucode files cannot be directly passed on to the link editor.

If the `-F O2` option is specified for a dual object input file which contains both object code and Ucode (see `-K dual`, page 59), the optimization based on the Ucode is performed first, and is followed by a linkage run. The object code is linked by processing the dual object file further without the optimization option (see page 97 for details).

The inline expansion of user functions can be suppressed with the `-F no_inlining` option.
If the option `-F i`,*file* is specified, the inline generator evaluates a control file named *file*, which contains the names of functions that must always or never be expanded inline (see page 57).
`-F O2` cannot be specified in combination with the link editor option `-G` (see page 66) or the debugger option `-g` (see page 59).

`-F O3`

Enables the optimization measures for `-F O2` as well as interprocedural register allocation.
All processing facilities and restrictions described for the `-F O2` option above are also applicable here. Note that the objects generated with `-F O3` cannot be linked dynamically, which means that the option `-d n` must be specified at link time (`-d y` for dynamic linking is the default setting).

`-F O4` [*option*]
`-F O5` [*option*]

The `-F O4` option enables feedback optimization (i.e. feedback-directed function inlining, branch elimination, procedure positioning, procedure splitting, feedback-directed register allocation, and feedback-directed instruction scheduling) as well as all the optimization measures at the `-F O2` level. The processing facilities and restrictions described for the `-F O2` option are also applicable here.

`-F O5` enables feedback optimization (see `-F O4` above) and all the optimization measures at the `-F O3` level. The processing facilities and descriptions described for the `-F O3` option are likewise applicable here.

Standard optimization process with no additional options [option] specified

By default, i.e. if no additional *option* is specified, the compiler looks for profiling raw data files with the name *PID.executable_file* in the `PROF` subdirectory of the current working directory. These are files that were created earlier on running the profiling executable (see `-q f` on page 62). The compiler extracts the profiling information from the individual profiling raw data files and combines this information in a profiling summary file named *executable_file*`.summary`, which it then places in the profiling directory *$PWD*/`PROF`. This profiling summary file is subsequently evaluated during the individual optimization phases. If no profiling raw data files are available, the compiler aborts with an error message.

Optimization process with additional options [option] specified

The additional *option* can be used to individually turn off or control the feedback-directed optimization measures. It can also be used to specify profiling raw data files and directories with different names (as opposed to the default names). The following values can be specified for *option*:

`-F default_inlining`
Disables the use of feedback information for function inlining, i.e. the optimization is the same as for the `-F O2` and `-F O3` options.

`-F no_inlining`
Disables function inlining.

`-F no_positioning`
Disables procedure positioning.

`-F no_splitting`
Disables procedure splitting.

`-F no_br_elimination`
Disables branch elimination as well as procedure splitting.

`-F no_feedback_uopt`
Disables the use of feedback information for register allocation, i.e. performs the same optimization as for the `-F O3` option.

`-F default_xbb`
Disables the use of feedback information for cross basic block instruction scheduling (see also `-F noxbb` on page 57).

`-F select_ucode,`*n*
This option is used to reduce the compilation time when processing dual object files. It restricts optimization based on the Ucode to modules with the highest feedback weight. The feedback weight is calculated from the feedback information as a percentage of the overall runtime taken up by a module. The remaining modules are simply linked into the program on the basis of the object code. *n* is an integer (where 0 <= *n* <= 100) that indicates the overall feedback weight in percent. On exceeding this value, no further Ucode optimization is performed. Practical experience has shown that a value of *n*=98 is appropriate here. It should also be noted that the object code of the dual object file is not profiled (see also the option combination `-c -K dual -q f` on page 30).

`-F profdata,'`*pattern*`'`
This option instructs the compiler to search for profiling raw data files that match the specified pattern. The pattern, which must be entered within single quotes in the form '`*.`*name*', causes the compiler to create the profiling summary file from profiling raw data files named *PID*`.`*name* instead of *PID*`.`*executable_file*.

`-F profdir,`*dir*
This option can be used to instruct the compiler to look for the individual profiling raw data files in the named directory *dir* and to place the profiling summary file in that directory.
If this option is omitted, the compiler will look for and save profiling files in the `PROF` subdirectory of the current working directory.

`-F feedback_summary,`*name*
Instructs the compiler to use the profiling summary file of the specified *name* for feedback optimization instead of creating its own profiling summary file.

`-F X`
`-F X4`
These options provide the best optimization in many cases. They are equivalent to a combination of options that are suitable for R4000-based (`-F X`) and R10000-based (`-F X4`) systems. Please check the latest information available for these options in the Release Notice, since the combination of options listed below may have changed.

`-F X` (for R4000)
Is currently equivalent to the following combination of options:

```
-K mips2
-K old
-F O3
-F I
-F Olimit,2000
-F G8
-F loopunroll,8
-d n
```

`-F X4` (for R10000)
Is currently equivalent to the following combination of options:

```
-K mips4
-K old
-F O3
-F I
-F Olimit,3000
-F G32
-F loopunroll,8
-F unrolllimit,2000
-F hw_br_predict
-B do_jmpopt
-d n
```

Other optimization options

`-F afep,`*n*
The compiler aligns function entry points to 2 the power of *n* bytes. The default setting is *n*=2, which means that the function entries are aligned on a word boundary. In the case of programs that contain several subroutines, performance can be improved significantly if the function entry points are aligned on more than one word. A value of *n*=5 or *n*=6 is recommended in such cases.

`-F big_got`
Causes the compiler to use the extended GOT code model when generating PIC code. This model allows the addressing of more than 16 Kbytes of global data in shared objects.

`-F Blimit,`*n*
This option is provided only for cases when the default value of *n* = 8 (number of 128-bit blocks) is not sufficient for optimization (uopt) and needs to be increased. If this is required, the compiler will issue a corresponding error message at compilation.

`-F G`*n*
This option can be used to change the default behavior of the `-K old` and `-K no_pic` (see page 60) options.
The option specifies to what length (*n*) global data with a 16-bit address offset should be addressed via the gp register. A maximum of 64 Kbytes of global data can be addressed by this efficient method. If this option is specified, programs which address more than 64 Kbytes of global data up to a length of *n* bytes via the gp register cannot be linked, and thus result in a linkage error. You will then need to recompile such

programs by specifying a lower value for *n*.
If the `-K old` option is used, global data is not addressed via the gp register (*n*=0) by default.
If the option `-K no_pic` is used, the option `-F G8` (gp addressing of global data up to a length of 8 bytes) is set implicitly.

`-F hw_br_predict`

This option is only meaningful in combination with `-K mips4` (see page 59).
It suppresses the generation of "branch-likely" instructions and causes the dynamic R10000 hardware branch prediction to be used instead. This option is set internally at the `-F X4` optimization level (see page 54).

`-F I`[*name*]

This option causes the C library function *name* to be generated inline, thus saving the overhead for the function call and return. This optimization is performed by the compiler frontend and can also be specified in combination with the debugging option `-g`. However, note that no breakpoints can be set on inlined functions when subsequently debugging with DBX.

Math functions that are generated inline are optimized for speed and are therefore not fully X/OPEN conformant. Due to performance reasons, it is not possible to supply the error variable `errno` in `string` and `memory` functions and `HUGE_VAL` for math functions as required by the XPG4 Standard.

The following C library functions can be inlined:

Name	Notes
`abs`	Only the integer parameters
`fabs`	Only the float/double parameters
`sqrt`	Only the float/double parameters
`alloca`	Only the integer parameters
`memset`	The parameter `size_t` *n* must be a numeric constant < 1024
`memcpy`	
`memmove`	

name not specified:
Turns on inlining for the maximum possible number of functions (including all C library functions listed above).

`-F i,`*file*

This option controls the inline expansion of user functions when using the `-F O`*x* (*x* = 2, 3, 4 or 5) options for optimization. *file* is a text file that contains lines in the form +*function_name* and −*function_name* to indicate the names of functions that must always (+) or never (−) the generated inline.
The + or − must be entered in the first column, and the function name must begin in the second column.
This option has no effect on the handling of C++-specific inline functions.

`-F inline_limit,`*n*

This option is used for optimization with the `-F O`*x* (*x* = 2, 3, 4 or 5) options. It restricts inlining of user functions to a basic block when the specified basic block size (*n*) is exceeded. *n*=500 is the default setting.
This option has no effect on the handling of C++-specific inline functions.

`-F loopunroll,`*n*

This option controls loop unrolling. The value *n*, which you specify, instructs the optimizer (uopt) how often (i.e. the maximum number of times) the body of a loop may be repeated (or "unrolled"). The default setting is *n*=4. If desired, loop unrolling can be suppressed by specifying a limit of *n*=0. See also option `-F unrolllimit` (page 58).

`-F no_inlining`

Suppresses the inlining of user functions during optimization with the `-F O`*x* (*x* = 2, 3, 4 or 5) options.
In the C++ modes, this option will also suppress the expansion of C++-specific inline functions, even if `-F O`*x* was not specified to turn on optimization.

`-F noxbb`

This option disables cross-basic block instruction scheduling, i.e. restricts it to within a basic block (which means that jump instructions are not taken into account).

`-F Olimit,`*n*

Restricts global optimization (by uopt) to functions that do not consist of more than *n* basic blocks. The default settings are *n*=1000 for the `-O` and `-F O`*x* (*x* = 2, 3, 4 or 5) optimization levels, *n*=2000 for `-F X`, and *n*=3000 for `-F X4`. If a function is not optimized due to the default or explicitly set value for *n*, the compiler issues a warning (with the value required for the optimization).
This warning can also occur when compiling a large C++ program which contains C++-specific inline functions and which uses C++ exception handling (usually as a result of the numerous temporary variable generated by the compiler). This problem can be solved by using the `-F U` option (see page 58).

`-F sortedges`
With this option, functions are selected for inlining during optimization with the `-F O`*x* (*x* = 2, 3, 4 or 5) options in the order of an increasing storage space to runtime ratio. The option is most effective with feedback optimization measures.

`-F space_time,`*n*
Restricts function inlining during optimization with the `-F O`*x* (*x* = 2, 3, 4 or 5) options to only those functions whose estimated ratio of code expansion to time savings is < *n*. The default setting is *n*=3.

`-F U`
Eliminates superfluous code, which typically consists of assignments to variables that are no longer needed in subsequent program runs, or variables whose values are already known at compile time and can therefore be replaced by constants. This optimization is performed on the compiler intermediate language level (ULS).
`-F U` cannot be specified in combination with the debugging option `-g` (see page 59).

`-F ucode`
Causes Ucode files and dual object files to be linked into an executable file on the basis of the Ucode, without any optimization. This option is implicitly used when generating a profiling executable with `-q f` if the input source is a Ucode object file or a dual object file (without profiling object code).

`-F unrolllimit,`*n*
This option controls loop unrolling; *n* specifies the maximum number of instructions for the unrolled loop. The default value is *n*=320. In other words, if unrolling the body of the loop would involve more than *n* instructions, no loop unrolling occurs. See also the option `-F loopunroll` (page 57).

`-K ucode_libraries`
`-K no_ucode_libraries`
If `-K ucode_libraries` is specified, the directory `/usr/ccs/lib/libu` is searched for libraries before the directory `/usr/ccs/lib`. This directory contains system libraries (e.g. the standard C library `libc.a`) in Ucode format and thus enables even the library functions to be optimized on the basis of the Ucode.
`-K no_ucode_libraries` is the default setting.

3.2.8 Options to control object generation

`-g`

The compiler generates additional information (such as line numbers, type information, etc.) for the symbolic debugger DBX as well as line numbers for `dis`.
If the `-g` option is specified in combination with one of the following optimization options, the compiler will issue a warning and turn off the optimization:
`-F U`, `-O`, `-F O`*x* (*x* = 2, 3, 4 or 5).

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control object generation:

`dual`
`no_dual`

If the `-K dual` option is specified in combination with `-c`, the compiler will generate "dual object files" under the name *file*`.o`. A dual object file contains object code in ELF format as well as Ucode (see also page 97). The object code of the dual object file created with `-K dual` can be optimized with the `-O`, `-F U` and `-FI` options and can be processed further (i.e. linked) like any conventional ELF object. The Ucode of a dual object file can be processed further in another compiler run with the `-F O`*x* (see page 51) options specified. Note that dual object files cannot be generated from Assembler source code (.s files).
`-K no_dual` is the default setting.

`mips1`
`mips2`
`mips3`
`mips4`

`-K mips1`
With this option, the compiler generates instructions for the MIPS 1 RISC architecture (which corresponds to an R3000 processor). If the option `-K bit32` is specified, `-K mips1` is the default; however, if the option `-K lp64` (64-bit data model) is specified in combination with `-K mips1`, then `-K mips1` is ignored, and `-K mips3` is assumed.

`-K mips2`
The compiler generates instructions for the MIPS 2 RISC architecture (which corresponds to an R6000 processor or an R4000 processor without 64-bit instructions). `-K mips2` is the default setting with the `-F X` option, but is ignored if specified in combination with `-K lp64` (64-bit data model). In the latter case, `-K mips3` is automatically assumed.

`-K mips3`
The compiler generates instructions for the MIPS 3 RISC architecture (which corresponds to an R4000 processor). `-K mips3` is the default setting with the `-K lp64` option. When the option `-K lp64` is specified, 64-bit instructions are also generated for R4000. No 64-bit instructions are generated in the default case (i.e. with the option `-K bit32` specified); however other R4000-specific instructions are used.

`-K mips4`
The compiler generates instructions for the MIPS 4 RISC architecture (which corresponds to an R10000 processor). `-K mips4` is the default setting with the `-F X4` option. If the option `-K lp64` is specified, 64-bit instructions are also generated for R10000. No 64-bit instructions are generated in the default case (i.e. with the option `-K bit32` specified); however other R10000-specific instructions are used.

`pic`
`no_pic`
`old`
`selpic`

`-K pic`
The option `-K pic`, which is set by default, results in the generation of ABI-conformant position-independent code (PIC).
Since global data is accessed indirectly in this case, a distinct performance degradation can be noted when accessing global data. The generated PIC code can be linked with shared objects, but only dynamically.

`-K no_pic`
The `-K no_pic` option can be used to select a more efficient method of addressing global data via the gp register, provided no ABI conformance is required.
`-K no_pic` has the same effect as combining the options `-K old` and `-F G8`. In other words, no ABI-conformant PIC code is generated, but global data that is ≤ 8 bytes in length can be addressed more efficiently by using the gp register. The default length of 8 bytes for the global data to be addressed via the gp register can be changed by explicitly specifying the `-F G`*n* option (see page 55).
Note that the objects generated with this option can only be linked statically and that the nopic versions of the standard libraries are automatically used
(e.g. `/usr/ccs/lib/libnopic/libc.a`).

`-K old`
The `-K old` option resembles the option `-K no_pic` to the extent that no ABI-conformant PIC code is generated; however, in contrast to `-K no_pic`, the objects generated with `-K old` can be linked with PIC objects and system libraries.
The option `-F G`*n* can be used to turn on addressing via the gp register for global data with a length ≤ *n* (see page 55).
The objects can only be linked statically.

`-K selpic`

This option is only useful in combination with the optimization options `-F O`*x* (where *x* = 2, 3, 4 or 5). It causes PIC code in dynamically executable files to be replaced by faster non-PIC code (`-K old`).
The following optimization measures are performed:

1. Function calls that use the global offset table (GOT) are replaced by direct jumps (`jal` instructions).
2. Non-PIC addressing is used to access data that is defined in the executable file.
3. Instructions to save and restore gp register are removed from the function prologs and epilogs.

`roconst`
`no_roconst`

These arguments specify the program section where constants that were declared with the type attribute `const` are to be stored.
`-K roconst` is the default setting, which causes the constants to be placed in the read-only section. Consequently, the constants cannot be overwritten, even if the `const` attribute is removed with a `cast` operator.

Warning: This only applies to global or local `static` constants. Local `auto const` variables cannot be stored in the read-only section.

If `-K no_roconst` is specified, the constants are not placed in the read-only section and can therefore be overwritten by other values if the `const` attribute is removed by means of a `cast` operator.

The `-K roconst` option is not accepted in a compilation run with `-K no_pic`.

`rostr`
`no_rostr`

These arguments specify the program section where string literals are to be stored.
If `-K rostr` is specified, string literals are stored in the read-only section and cannot be overwritten.
`-K norostr` is the default setting, which means that string literals are not stored in the read-only section and can therefore be overwritten.

The `-K rostr` option is not accepted in a compilation run with `-K no_pic`.

`thread`
`no_thread`

If `-K thread` is specified, the program may use "multi-threading" facilities, e.g. threaded C functions and `pthread` interfaces. Note that the required header files (see option `-I` on page 37) and thread libraries (see option `-l` on page 68) are automatically searched and linked in the correct order in threaded mode.
Note also that DCE (Reliant UNIX) V2.0 is a prerequisite for these arguments.
`-K no_thread` is the default setting.

`-q p`
`-p`
`-q l`
`-q f`

These options control the generation of profiling information.

`-q p` or `-p`
The compiler generates additional code for each function in order to count how many times a function is called. When you subsequently run the compiled and linked program, a profiling raw data file named `mon.out` containing the counter values is generated on successful completion of the program. Note that the program must be linked statically for this purpose.
The profiling raw data file can be used with the `prof` command to create a runtime profile of the program. `prof` returns the CPU time and call graph information for all functions of the program that were compiled with the `-q p` option. You will find detailed information on the `prof` command in the "Programmer‘s Reference Manual".
These options are not supported in combination with `-K no_pic`.

`-q l`
The compiler generates code for each source code line in order to enable line profiling. On completion of the program, a file named *program_name*`.cnt`, which can be analysed with `lprof`, is created and placed in the current directory. More information on the `lprof` command can be found in the "Programmer‘s Reference Manual".
These options are not supported in combination with `-K no_pic`.

`-q f` [`-F profdata,'`*pattern*`'`] [`-F profdir,`*dir*]
For compatibility reasons, `-q feedback` may also be specified instead of `-q f`.
This option instructs the compiler to create a "profiling executable" without optimization. This profiling executable is assigned the name `a.out` or the name specified with the `-o` option. Whenever the profiling executable is run, it generates a profiling raw data file containing measurement data and statistics concerning the program flow and function calls (i.e. the call graph and flow graph information). These profiling raw data files are used for feedback optimization in a later compiler run in which the optimized version of the executable file is generated (see options `-F O4` and `-F O5`).

Standard flow of processing without the additional options -F profdata and -F profdir

If required, the compiler creates a profiling directory named PROF as a subdirectory of the current working directory. This standard profiling directory is also referred to as *$PWD*/PROF in the discussion below.

The compiler then places a copy in of the generated profiling executable in the profiling directory. This copy is assigned the name *executable_file*.fb. The copy is required internally for subsequent feedback optimization and must therefore not be renamed or deleted. Whenever the profiling executable is run, a profiling raw data file is created and placed in the same profiling directory under the default name *PID.executable_file*, where *PID* is the process ID and *executable_file* is either a.out or the name defined with the -o option.

If any profiling raw data files that match the pattern *PID.executable_file* are present in the profiling directory at the time of creating the profiling executable, such files are deleted by the compiler.

Flow of processing with the additional options -F profdata and -F profdir specified

If desired, you can use the option -F profdir*,dir* to specify the pathname of some other profiling directory in which the copy of the profiling executable (.fb file), and the profiling raw data files which are generated on running that executable, are to be stored. By default, this profiling directory is named *$PWD*/PROF (see above).

If the option -F profdata,'*pattern*' is specified, all profiling raw data files that match the specified pattern are deleted by the compiler when the profiling executable is created. The specified pattern, which must be enclosed within single quotes in the form '*.*name*', instructs the compiler to delete the profiling raw data files that match *PID*.*name* instead of those which match *PID*.*executable_file*.

Input sources for the generation of profiling executables

The input sources for creating profiling executables with the -q f option may be C/C++ source files and object files (file.o).
The object files involved may consist of only object code, only Ucode (Ucode file) or both (dual object file).

Object files that contain only object code in ELF format are linked into a profiling executable; however, only those program sections which were generated in an earlier compilation run with the -q f option in combination with -c are profiled for the generation of profiling information.

The object code within a dual object file will have already been profiled for generating profiling information if it was created in an earlier compilation run with the options -q f, -c and -K dual.

In the case of Ucode files or dual object files in which the object code is not profiled, the option `-F ucode` is set implicitly. In other words, these program sections are profiled and linked on the basis of the Ucode without any optimization.

When source files are precompiled with `-q f -c`, the additional options `-F profdir` and `-F profdata` are ignored.

Warning:
No profiling executables can be generated from shared objects (libraries).

Feedback-directed optimization is described in detail starting on page 103.

`-Q y`
`-Q n`

This option controls whether or not compiler identification information is included in the output files generated in the individual compilation phases (see "Options to select compilation phases" on page 30). `-Q y` is the default setting.
If `-Q n` is specified, no identification information is included in the output files.
The compiler identification information included in object files can be queried with the following command:
`dump -v -s -n .comment` *file*`.o`

3.2.9 Link editor options

All options in the `cc/c89/CC` command that are not known to the compiler, i.e. options which begin with an unrecognized letter after the hyphen "-", are passed through to the link editor (see the `ld` command in the "Programmer‘s Reference Manual").

Some of the options intended for the `ld` command could also be specified in the `cc/c89/CC` call as follows:

– The `ld` options `-B`, `-d`, `-e`, `-G`, `-h`, `-L`, `-r`, `-s` and `-u` may be specified directly.

– The `-o` and `-V` options may also be specified directly; however, these options are also significant for other compilation phases. Consequently, if they are specified in the `cc/c89/CC` call, they affect all the executed phases for which they are defined.

– The option `-W l` can be used to pass any `ld` option to the link editor. All options specified here are only valid for the link editor.

`-B dynamic`
`-B static`

The `-B dynamic` option instructs the link editor `ld` to search the archive specified with `-l` *archive_code* for dynamic libraries (`lib`*archive_code*`.so`) and, if the search is not successful, for static libraries (`lib`*archive_code*`.a`) as of the position where the option is specified on the command line. This continues until the end of the command line or until a `-B static` option is encountered.

The `-B static` option instructs the link editor `ld` to search the archive specified with `-l` *archive_code* only for static libraries (`lib`*archive_code*`.a`) as of the position where it appears on the command line. This continues until the end of the command line or until a `-B dynamic` option is encountered.

`-B dynamic` and `-B static` fall under the "operands" category and may therefore also be specified after ending the input of options with `--` (see also the section on "Operands" on page 23).

`-B symbolic`

Specifies how unresolved references are to be satisfied at the time of dynamic linking. If an object module contains definitions for global symbols, `ld` satisfies the unresolved references to these symbols at the time of generating the shared object module. This ensures that the object-specific definition for a symbol is used when available. Normally, references to global symbols in a shared object module are resolved only at runtime, even if the definitions are available. This could lead to the use of a definition from a previously specified executable or object module instead of the object-specific definitions for a symbol. You can prevent this from occurring by specifying the `-B symbolic` option.

`-B do_jmpopt`

This option is relevant for branch optimization and is only meaningful in combination with `-K mips4` (see page 59), since it enables R10000-specific filling of the branch delay slot. `-B do_jmpopt` is used internally at the `-F X4` optimization level (see page 54).

`-d y`
`-d n`

Global setting of `ld` for dynamic or static linking:

`-d y` Dynamic linking (default).
The executable program contains references to shared libraries which are linked dynamically at runtime. The dynamically linkable version of the standard libraries are used (e.g. `libc.so`).

`-d n` Static linking.
All objects are linked statically into an executable program using the statically-linkable version of the standard libraries (e.g. `libc.a`).

`-e` *name*

Sets the address of the symbol *name* as the entry point for the generated executable program.

`-G`

The `-G` option instructs the link editor to generate a shared object and causes any other optimization options specified with `-F O`*x* (where *x* = 2, 3, 4, or 5) or the link editor option `-d n` to be ignored without a warning. Standard libraries are not automatically linked, and unresolved references do not lead to an error message. The shared object is assigned the name `a.out` or the name specified with the `-o` option.

`-h` *name*

The `-h` option can be used when creating a shared object with option `-G` to specify a name with which that object is referenced at the time of execution. This name is inserted in the shared object as a fixed reference for the dynamic linker/loader (`ldd`) and is entered in the executable program (`.dynamic section`) at link time. The dynamic linker/loader then searches for *name* when dynamically loading the object at runtime. If the `-h` option is omitted, the name specified at link time is inserted into the executable program.
The name that is assigned with `-h` and entered into the program as a name reference for the dynamic linker/loader must not be confused with the file name of the library in which the shared object created with `-G` is placed. This library is assigned the name `a.out` or the name that was specified with the `-o` option.
The option `-h` *name* is effective only in combination with option `-G`.

`-K` *arg1*[,*arg2*...]

General input rules for the `-K` option can be found on page 23.
The following entries are possible as *arg* arguments to control the link editor:

`link_stdlibs`
`no_link_stdlibs`

`-K link_stdlibs` is the default setting and causes certain standard libraries to be automatically linked. This means that the corresponding `-l` options are automatically set for these libraries:

1. Only with the option `-K thread` (see page 62)

   `-l cma`

2. Only with the `CC` command

   `-l Cstd` and `-l Crun` in ANSI C++ modes
   `-l C` and `-l CFrun` in Cfront C++ mode

3. Only with the option `-q p` (generation of a profiling raw data file for further processing with `prof`; see page 62)

   `-l prof`
   `-l elf`
   `-l m`

4. Only with the option `-K bit32` together with `-K longlong`

   `-l dw`

5. Always

   `-l c`

If `-K no_link_stdlibs` is specified, the above libraries are not linked automatically. This is useful, for example, if the `-y` option is used to stop the compiler run after the prelinker (see page 32) or if the `-r` option is used to generate a prelinked object file (see page 69).
`-K no_link_stdlibs` is set automatically when shared objects are generated by using the `-G` option (see page 66).

`link_startups`
`no_link_startups`

`-K link_startups` is the default setting and and causes startup object files to be automatically linked.
If `-K no_link_startups` is specified, some startup object files are not automatically linked. This option is useful in combination with the option
`-K no_link_stdlibs`.

`-l` *archive_code*

Instructs `ld` to use the library specified with *archive_code* for linking. By default, `ld` searches for the library by checking the following directories in the given order:

1. The directories specified with `-L`
2. Either the directories specified with the option `-Y P` (see page 70) or the following standard directories.

Standard directories that are searched last

The searched directories for the default 32-bit data model (see option `-K bit32`) and for the 64-bit data model (see option `-K lp64`) are .../`lib` and .../`lib64s`, respectively.

a) `/opt/CDS++/lib`[`64s`]`/thread`
`/opt/thread/lib`[`64s`]
`/usr/ccs/lib`[`64s`]`/thread`
`/usr/lib`[`64s`]`/thread`

if the option `-K thread` is specified

b) `/opt/CDS++/lib`[`64s`] only with the `CC` command in all C++ modes

c) Special directories, depending on certain options

`/usr/ccs/lib`[`64s`]`/libu` if the option `-K ucode_libraries` is specified

`/usr/ccs/lib`[`64s`]`/libnopic` if the option `-K no_pic`

`/usr/ccs/lib`[`64s`]`/libp` if the option `-q p` or `-q f` is specified

`/usr/lib`[`64s`]`/libp` if the option `-q p` or `-q f` is specified

d) `/usr/ccs/lib`[`64s`]

e) `/usr/lib`[`64s`]

If dynamic linking is selected, `ld` will first search for the dynamic version of the library named `lib`*archive_code*`.so` and use that library; otherwise, it will look for only the static version `lib`*archive_code*`.a` and link that static library.
Instead of an absolute file name for the desired library, you can also specify just the main library component with *archive_code*.

`-l` *archive_code* falls under the "operands" category and may therefore also be specified after ending the input of options with `--` (see also the section on "Operands" on page 23).

`-L` *dir*

dir specifies an additional directory to be searched by the link editor for the libraries specified with `-l` options. A directory specified with `-L` is searched for libraries before the directories of the standard libraries. The search order followed by the link editor is determined by the order in which the `-L` options are listed on the command line.
This option falls under the "operands" category only for the `cc` and `CC` commands and may therefore be specified after ending the input of options with `--` (see the section on "Operands" on page 23) only when using those commands.

`-r`

This option can be used to prelink multiple object files into a single object file. Since such object files can only be linked statically, the option for static linkage `-d n` is assumed implicitly without a warning if it hasn‘t been specified (`-d y` for dynamic linkage is the default). Prelinked object files are not executable and contain relocation information that is required for a subsequent linkage run. If the standard libraries are not to be linked, the option `-K no_link_stdlibs` must also be set.
Unresolved references do not result in an error message.
The prelinked object file is assigned the name `a.out` or the name specified with the `-o` option. It can be processed further in a useful context (i.e. linked) only if the name contains the suffix `.o` or a suffix that was designed with the `-Y F` option (see page 28).

`-s`

Symbol table information is stripped from the output file. The sections with additional information for troubleshooting and with line numbers and associated offset information are also removed.
If you want to test the program with the symbolic debugger DBX, you should not specify this option together with option `-g`.

`-u` *name*

This option causes *name* to be entered into the symbol table as an undefined symbol. It can be used to force the loading of modules from libraries even if no explicit references to such modules are present in the program.

`-W l`,*ld_option*[,*ld_option*...]

This option allows you to pass options to the link editor `ld`. *ld_option* can be any link editor option, but must be enclosed within single quotes if the link editor option contains blanks.

`-Y P,`*dir1*`[:`*dir2*...]
Instructs the link editor to search for libraries in the directories specified with *dir* last. Without this option, the last directories to be searched are the standard directories listed for the `-l` option under point 2 (see page 68).

`-Y S,`*dir*
Instructs the link editor to take the startup object files from the specified directory *dir*. The startup files are normally obtained from the directory `/usr/ccs/lib`.
The modules (`*crt*.o`) involved are generally required for some of the initialization and completion tasks of a program.

3.2.10 Options to control message output

`-R limit,`*n*

This option defines the maximum number of errors to be tolerated by the compiler before aborting the compilation run. Notes and warnings are counted separately. The default value is *n* = 100. If *n* = 0, the compiler will attempt to continue compiling as long as possible, regardless of the number of errors that have occurred.

`-R min_weight,`*min_weight*

This option defines the minimum error weight (i.e. severity code) as of which diagnostic messages from the compiler are to be output to the standard error stderr.
`-R min_weight,warning` is the default setting.
The following entries are possible for *min_weight*:

`notes`	All messages are output, i.e. even the notes.
`warnings`	The output of notes is suppressed (default).
`errors`	The output of notes and warnings is suppressed.
`fatals`	The output of notes, warnings and errors is suppressed.

`-R msg_id`
`-R no_msg_id`

`-R msg_id` is the default setting and causes the message ID to be output together with the diagnostic messages of the compiler.
If `-R no_msg_id` is specified, the message ID is not included with the message.

`-R msg_wrap`
`-R no_msg_wrap`

`-R no_msg_wrap` is the default setting, which causes diagnostic messages to be output without line wrapping.
If `-R msg_wrap` is specified, message lines are wrapped after 80 characters.

`-R note,`*msgid*,[*msgid*...]
`-R warning,` *msgid*,[*msgid*...]
`-R error,`*msgid*,[*msgid*...]

These options can be used to change the default severity code of diagnostic messages. *msgid* is the corresponding message number. The severity code for fatal errors cannot be changed. This also applies to errors, unless they have been explicitly marked in the original message with an asterisk: `[*error]`. Depending on the language mode or the position in the code, the same message ID *msgid* can have a different severity code (warning or error).

`-R show_column`
`-R no_show_column`

This option determines whether the diagnostic messages of the compiler are generated in short or long form.
`-R show_column` is the default setting, which means that the original source program line is shown with the error location marked (with `^`) in addition to the diagnostic message itself.
If `-R no_show_column` is specified, the marked source program line is not output.

`-R strict_errors`
`-R strict_warnings`

This option is only useful only in the strict ANSI C/C++ modes (`-X c`, `-X e`).
`-K strict_warnings` is the default, which means that warnings are issued for language constructs that deviate from the ANSI/ISO standard, but do not represent a serious violation of the semantic rules defined therein (e.g. implementation-dependent language extensions; see chapter 7).
If `-K strict_errors` is specified, such cases are treated as errors with corresponding messages.
Serious violations automatically lead to errors.

`-R suppress,`*msgid*,[*msgid*...]

Suppresses the output of the message with the message ID *msgid*. Some messages (e.g. fatal errors) cannot be suppressed.

`-R use_before_set`
`-R no_use_before_set`

`-R use_before_set` is the default setting and causes warnings to be issued if local `auto` variables are used in the program before being assigned a value.
If `-R no_use_before_set` is specified, the output of such warnings is suppressed.

`-v`

The output of messages with this option is the same as for the option combination `-R min_weight,notes` and `-K verbose`.

`-w`

This option is a synonym for `-R min_weight,errors`.

3.2.11 Options to output listings and CIF information

`-N output`,*file*

This option allows you to specify the name of an output file as an alternative to the listing options `-N shortsource` or `-N shortxref`.
If you specify a *file* with `-N output` and also with the listing options, the entry with the listing options is used as the output file.

`-N shortsource`[,*lfile*]

The compiler generates a source/error listing in a simple unedited format containing the source program lines of the primary source file, the header files used in it, and the diagnostic messages of the compiler.
If *lfile* is omitted, a source/error listing is generated for each compiled source file and placed in a corresponding file named *sourcefile*`.lst`.
lfile can be used to define another file name for the output of the source/error listing. Only one source file can be compiled in this case. If multiple source files are specified in the compiler call, the file *lfile* will contain only the source/error listing for the source file that was specified last in the command line.
The `-N output` option can be used as an alternative method of specifying a file name instead of *lfile*; however, there must always be only one source file for the source/error listing even in this case.

Each line in the list begins with a letter that identifies the line type:

`N` Normal source program line

`X` Expanded form of a normal source program line, typically containing expanded macro calls (see example) or alternative notations (trigraph sequences).

```
N #define p(x) (x + 2 * x + 4 * argc / 5)
                ...
N return p(argc);
X return (argc + 2 * argc + 4 * argc / 5);
```

`S` Source program line that is skipped due to an `#if` condition. Lines containing statements to end such branches (`#else`, `#elif`, `#endif`) are marked as normal source program lines (`N`).

`L` Indicates a change in the source program position, using the following format:

`L` *line_number* `"`*file_name*`"` [*key*]

The first `L` line in the source listing identifies the primary source file and does not include any *key* in this case:
`L 1 "`*sourcefile_name*`"`

`L` lines are also output without a *key* for `#line` directives.

key can have a value of 1 or 2, where 1 indicates the start of a header file *file_name*, and 2 stands for the end of that header file.

`R, W, E, C`
Identifies a diagnostic message, using the following format:

S "*file_name*" *line_number column_number message_text*

S stands for the **S**everity code:
`R` (**R**emark = Notes)
`W` (**W**arning)
`E` (**E**rror)
`C` (**C**atastrophic error = fatal or internal error)

Errors that are not associated with the source program (e.g. errors in the command line) or with a particular source program line/column are indicated with an empty string `""` as the *file_name*, and a *line_number* and *column_number* with a value of 0.

The continuation lines of a diagnostic message are essentially output in the same format as the first diagnostic line (*file_name*, *line_number*, etc.), except for the fact that the key for the severity code is entered as a lowercase letter (`r`, `w`, `e`, `c`) in the continuation lines.

`-N shortxref`[,*xfile*]
The compiler generates a cross-reference listing.
This cross-reference listing is output as described for the source/error listings (see the notes under `-N shortsource` on page 73) to a file named *sourcefile*`.lst` or the file specified with *xfile* or with `-N output`.

Every reference to an identifier is documented in the cross-reference listing in a line with the following format:

symbol_id name X file_name line_number column_number

Every identifier *name* is assigned a unique number *symbol_id*.

X specifies the type of reference:

`D` (definition)
`d` (declaration)
`M` (modification)
`A` (address)
`U` (used)
`C` (changed) stands for "used" and "modified" in the same operation, e.g. an increment
`R` (reference) - for all other references
`E` (error) - for references that are illegal or indeterminate

-N cif[,[*file*][,*tool1*...]]
The compiler generates a Compiler Information File (CIF) containing data that is relevant for further processing with the specified tool and places this information in the named *file*.
If *file* is not specified, the CIF is written to a file named *sourcefile*.cif.
The currently supported tool is the Static Analyser, so static_analyser is the default for *tool1*.

3.3 Files

file.c — C (cc, c89) or C++ (CC) source file before the preprocessor run

file.i — C (cc, c89) or C++ (CC) source file after the preprocessor run

file.C /.cpp/.CPP/.cxx/.CXX/.cc/.CC/.c++/.C++
C++ source file before the preprocessor run

file.I — C++ source file after the preprocessor run

file.s — Assembler source file

file.o — Object file with object code, Ucode (Ucode file) or both (dual object file)

file.a — Static library with object modules

file.so — Shared dynamic library with object modules

file.lst — Source, error, or cross-reference (xref) listing

file.cif — File with CIF data for further processing with the static analyzer

file.pch — Precompiled header file (PCH file)

file.o.ii — Information file for automatic template instantiation (used internally)

a.out — Executable file

mon.out — File with measurement data for further processing with prof

file.cnt — File with measurement data for further processing with lprof

file.mk — Preprocessor output file for further processing with make

executable_file.fb
Copy of the profiling executable for feedback optimization (used internally)

executable_file.summary
Profiling summary file for feedback optimization (used internally)

PID.executable_file
Profiling raw data file for feedback optimization (used internally)

/var/tmp/... — Intermediate files of the compilation run

/opt/CDS++/bin/...
/opt/CDS++/lib/...
/opt/CDS++/include/...
Default installation directories for the compiler components (bin), the C++ libraries and runtime system (lib), and the header files for the C++ libraries (include)

`/opt/lib/nls/msg/En/cds.cat`
`/opt/lib/nls/msg/De/cds.cat`
English (`En`) and German (`De`) message catalogs

`/usr/lib/locale/CDS++/LC_MESSAGES/cds.cat`
Message catalog that is used if the combination of `NLSPATH` and `LANG` results in a conflict

`/opt/CDS++/lib/messages`
English and German message texts as ASCII files

`/opt/CDS++/doc/`...
C++ language addendum and manuals in various formats (see the Release Notice for details)

3.4 Predefined preprocessor names

Some preprocessor macros and assertions are predefined, depending on which command and which options are entered when calling the compiler with `cc`, `c89` or `CC`.

Predefined preprocessor macros (defines)

`_BOOL`	Option `-K bool`
`__cplusplus`	In all C++ language modes: == 1 in Cfront C++ mode (`-X d`) == 2 in extended ANSI C++ mode (`-X w`) == 199504L in strict ANSI C++ mode (`-X e`)
`c_plusplus`	In all C++ language modes (`-X d|w|e`)
`__CFRONT_V3`	In Cfront C++ mode (`-X d`)
`host_mips`	Always set
`LANGUAGE_C`	The input file is a C or C++ source file
`_LANGUAGE_C`	The input file is a C or C++ source file
`_LONGLONG`	Option `-K longlong`
`__LP64__`	Option `-K lp64`
`mips`	Option `-K mips`*n*: == 1 for `-K mips1` == 2 for `-K mips2` == 3 for `-K mips3` == 4 for `-K mips4`
`__mips`	As for `mips`
`MIPSEB`	Always set
`__OLD_SPECIALIZATION_SNTAX`	== 1 with the option `-K old_specialization`
`sinix`	Always set
`__SIGNED_CHARS__`	Option `-K schar`
`SNI`	Always set
`__SNI_DCOSX`	Option `-K thread`
`__SNI_HOST_UNIX`	Always set

`__SNI_SVR4`	Option `-K thread`
`__SNI_TARG_UNIX`	Always set
`__SNI_THREAD_SUPPORT`	Option `-K thread`
`__STDC__`	Always set: == 0 in the K&R C (`-X t`), extended ANSI C (`-X a`), Cfront C++ (`-X d`) and extended ANSI C++ (`-X w`) modes == 1 in the strict ANSI C (`-X c`) and strict ANSI C++ (`-X e`) modes
`__STDC_VERSION__`	In all C language modes: == 0 in the K&R C (`-X t`) and extended ANSI C (`-X a`) modes == 199409L in strict ANSI C mode (`-X c`)
`unix`	in K&R C mode (`-X t`)
`_WCHAR_T`	Option `-K wchar_t_keyword`
`_XPG_IV`	`c89` command

Predefined preprocessor assertions

`cpu(mips)`	Always set
`cpu(r3000)`	Option `-K mips1`
`cpu(R3000)`	Option `-K mips1`
`cpu(r4000)`	Option `-K mips2` or `-K mips3`
`cpu(R4000)`	Option `-K mips2` or `-K mips3`
`cpu(r10000)`	Option `-K mips4`
`cpu(R10000)`	Option `-K mips4`
`compiler(CDS++)`	Always set
`compiler(CDS)`	In all C language modes
`data_model(bit32)`	Option `-K bit32` (default)
`data_model(lp64)`	Option `-K lp64`
`machine(mips)`	Always set
`system(sinix)`	Always set
`system(unix)`	Always set

3.5 Structure of compiler messages

If errors occur when compiling a source program, the compiler outputs diagnostic messages to the standard error stderr.

Format of a diagnostic message

---

file_name line_number : [*severity_code*] : *msgid message_text*

---

file_name
Name of the source file containing the invalid source code

line_number
Source program line in which the error occurred

[*severity_code*]

`[note]` Inconsequential errors, e.g. "ugly" or superfluous constructs that usually have no impact on subsequent program behavior.
Notes are not issued automatically unless the option `-R minweight,notes` is specified (`-R minweight,warnings` is the default setting).

`[warning]` Errors for which the compiler will generate a module, but which could lead to situations resulting in deviant program behavior.

`[error]`, `[*error]`
Errors for which no module is generated. The compiler will attempt to continue the compilation until the number of errors specified with the `-R limit,`*n* option is reached. Errors that can either be reduced to the severity of a "warning" with the `-R warning,`*msgid* option and warnings that were upgraded to the severity of an "error" with the `-R error,`*msgid* option are flagged with an asterisk.
See also the examples on page 81.

`[fatal]`, `[internal]`
Fatal errors that cause the compilation to be aborted immediately.

msgid
Message ID that uniquely identifies a diagnostic message.

message_text
Text of the error message (German or English)

If the option `-R show_column` is set (the default), the original source program line containing the error is output in addition to the diagnostic message, and the location of the error is flagged (with ^). If desired, the output of the flagged source program line can be suppressed with `-R no_show_column`.

Examples

```
$ cc -X c -R error,CFE1064 test.c

test.c     1: [*error]:   CFE1064 declaration does not declare anything
  struct {};
    ^

test.c     3: [error]:   CFE1077 this declaration has no storage class or
        type specifier
xxxxx;
^

user.h     3: [*error]:   CFE1054 too few arguments in macro invocation
  int i = A(3);
             ^
```

The error number CFE1064 was originally a warning, but was upgraded to the "error" class with the option `-R error,CFE1064`.
The class of errors that are not originally output with an asterisk (e.g. CFE1077 in this case) cannot be changed.
Error number CFE1054 is an example of an error that is generated in strict ANSI mode (`-X c`), but can be downgraded to the severity of a warning with `-R warning,CFE1054`.
The same error number is assigned the error class "warning" in extended ANSI mode (`-X a`).

Message catalogs and environment variables

The diagnostic messages of the compiler are supplied with English and German message texts in the two message catalogs `/opt/lib/nls/msg/En/cds.cat` (English) and `/opt/lib/nls/msg/De/cds.cat` (German), respectively. If the message texts appear in English, the environment variable `LANG` has been set to `En_US.ASCII`. If you need to have the message texts output in German, `LANG` must be assigned the value `De_DE.8859` and then exported:

```
$ LANG=De_DE.8859;export LANG
```

The message catalogs can be accessed without a problem if the value set for the environment variable `NLSPATH` on installing CDS++ has not been changed. In cases where the combination of the `LANG` and `NLSPATH` variables do not produce a valid path to the message catalogs, the compiler accesses the default catalog `/usr/lib/locale/CDS++/LC_MESSAGES/cds.cat`.

4 Precompiled header files

Programs in C and in C++ often require the same set of header files. These headers are inserted into the program by means of corresponding `#include` statements.

When compiling a source file, the CDS++ compiler can store the compiled code of the headers used in it in a separate file. This precompiled code can then be reused for the compilation of the same file or some other source file. The files which contain the precompiled code of headers are referred to below as “PCH” files. PCH is an abbreviation for **p**re**c**ompiled **h**eader.

The reuse of PCH files can speed up compilation in many cases, especially when the headers contain a lot of code (usually declarations) and when the primary source files which include them are relatively small. However, since PCH files take up a lot of disk space, they are most useful when several programs use the same PCH file. The prerequisites for this are listed in the section “Prerequisites for reusing PCH files” on page 86.

The generation and reuse of PCH files are controlled by means of compiler options:

`-Z pch` turns on the “automatic” PCH mode (see page 87).
`-Z create_pch` and `-Z use_pch` turn on the “manual” PCH mode (see page 89).

There are also some additional options and `#pragma` directives that can be used to control the handling of PCH files both in automatic and in manual PCH mode (see page 90).

The precompilation of headers is independent of the language mode and can thus be used for both C and C++ compilations.

If PCH files are to be generated or reused, only one source file may be compiled in each compiler call. If multiple source files are specified for a compilation and a PCH mode has been activated, the compilation is aborted with an error message.

4.1 Basic aspects of PCH file handling

When a PCH file is created on compiling a source file, it contains a snapshot of the compilation state for only the code that precedes the so-called “header stop point” (see “The header stop point” below).

In order to generate a PCH file, the primary source file and the headers included in it must satisfy specific conditions (see the section “Prerequisites for generating PCH files” on page 85 for details).

Apart from the precompiled code, the PCH file also contains some prefix information to determine whether it can be reused when compiling a source file. The prefix information of a given source file (which is temporarily generated internally) is compared with the prefix information of a PCH file by the compiler to check whether the latter is applicable to the current compilation. See the section “Prerequisites for reusing PCH files” on page 86.

The header stop point

The header stop point is the first token in the primary source file that does not belong to a preprocessor directive, but may also be specified directly with the preprocessor directive `#pragma hdrstop`. When generating a PCH file, the compiler writes the compiled code into the PCH file only up to the header stop point in the primary source file.

Examples for the header stop point

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

```
//b.C
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
```

In both cases above, the PCH file will contain the precompiled code of the headers `xxx.h` and `yyy.h`. In the first example, the header stop point is `int` (i.e. the first non-preprocessing token); in the second example, the directive `#pragma hdrstop`.

If the first non-preprocessing token or the `hdrstop` pragma appears within an `#if` block, the header stop point is the outermost enclosing `#if`.

Example

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

Here, the first non-preprocessing token is again `int`, but the header stop point is the start of the `#if` block containing it. The PCH file will contain the code of `xxx.h` and, conditionally, the definition of `YYY_H` as well as the code of `yyy.h`. It will not include the code of the `#if TEST` block.

Prerequisites for generating a PCH file

A PCH file will be produced only if the header stop point (see page 84) and the code preceding it (mainly, the header files themselves) meet certain requirements:

- The header stop point must be fully within the file scope of the primary source file.

  For example, a PCH file will not be created in the following cases:

  – The header stop point lies in a scope that begins in the header and ends in the primary source file.

  ```
  // xxx.h
  class A {

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

  – The header stop point is a declaration that begins in the header and is continued in the primary source file

  ```
  // yyy.h
  static

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

- No error should have occurred when compiling the header files preceding the the header stop. (**Note:** warnings and other diagnostic messages are not reproduced when the PCH file is reused.)
- No references to the 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.
- At least one declaration or definition, which typically also includes the definition of a name or macro with a `#define` directive, must appear in the header files. Otherwise, if the headers contain only preprocessor directives other than `#define`, no PCH file will be created.

Prerequisites for reusing PCH files

If a PCH file that was generated when compiling a given source file is to be reused for the compilation of the same source file or some other source file, the following specifications must match:

- the version of the compiler frontend, including the date and time the compiler was built
- the current directory at the time of compilation
- the command line options (e.g. language mode and frontend options) relevant for compilation and code generation
- the type and sequence of preprocessing directives in the primary source file, including `#include` directives
- the date and time of the header files specified in the `#include` directives

4.2 Automatic PCH mode (-Z pch)

The automatic processing of PCH files is enabled with the `-Z pch` option. In this mode, the compiler generates and uses PCH files in accordance with its own algorithm, which is explained below.

Precompiled headers generated in automatic PCH mode are always assigned names with the suffix `.pch`.
When the compiler creates a PCH file on compiling a source file, the name of the PCH file is constructed from the base name of the source file, with the suffix replaced by `.pch`. For example, the new PCH file generated for a source file `a.c` would thus be `a.pch`.
When the compiler reuses a PCH file, the base name of the reused PCH file need not be identical with that of the source file. For example, a PCH file named `a.pch` could also be used to compile a source file named `b.c`, provided the contents match.

Example

Consider the following two source files:

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

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

If no PCH file is present, and `a.C` is compiled with `-Z pch`, a PCH file named `a.pch` will be created. Then, when `b.C` is compiled or when `a.C` is recompiled, the PCH file `a.pch` will be reused in both cases, i.e. the precompiled code of the header `xxx.h` will be read.

By default, PCH files are written to or read from the current directory. If desired, some other directory may be specified by using the option `-Z pch_dir,`*dir*.

In some cases, more than one PCH file may be applicable to a given compilation. If so, the the largest PCH file, i.e. the one representing the most preprocessing directives from the primary source file, is used.
It is also possible that a PCH file may match, but not contain the entire code up to the header stop point. In this case, the code of the PCH file is read in first, and only the remaining code is then compiled.

Example

Let us assume that a primary source file `x.c` begins as follows:

```
#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 that both are applicable to the current compilation). Furthermore, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file named `x.pch` for all three headers is created.

Precompiled headers are considered obsolete by the compiler and deleted under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header, but is otherwise applicable to the current compilation.
- 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 to the current compilation (e.g. due to different command-line options).

4.3 Manual PCH mode (-Z create_pch, -Z use_pch)

The option `-Z create_pch,` *filename* specifies that a PCH file of the specified name should be created. If *filename* does not contain any directory components, the PCH file is placed in the current directory.

The option `-Z use_pch,`*filename* specifies that the named PCH file should be used for this compilation. If *filename* does not contain any directory components, the compiler will look for the PCH file in the current directory.

If the PCH file in invalid (i.e. if its prefix does not match that of the source file), a warning will be issued and the PCH file will not be used.
If the PCH file applies to the current compilation, but does not cover the entire code up to the header stop point in the primary source file, the precompiled code in the PCH file is used, and the remaining code is compiled.

Example

If the source file contains the following `#include` statements

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
   ...                // Start of Code
```

and the PCH file assigned with `-Z use_pch` contains only the code for `xxx.h` and `yyy.h`, the PCH file will be reused, and only the code in `zzz.h` will be compiled.

If the option `-Z pch_dir,`*dir* is specified in combination with `-Z create_pch,`*filename* or `-Z use_pch,`*filename*, the specified *filename* (possibly with directory components) will be appended to the directory name *dir*.

4.4 Other control possibilities

The following additional options can be used to control the handling of PCH files in both automatic and manual PCH mode.

- The pragma `#pragma hdrstop` can be inserted in the primary source file at a point before the first non-preprocessing token (header stop point) to specify where the generation or reuse of the PCH file is to end. Under normal circumstances, the header stop point is the first non-preprocessing token in the primary source file (see also page 84).

  Example

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

  Here, the precompiled header file will contain the precompiled code for `xxx.h` and `yyy.h`, but not for `zzz.h`.

- The pragma `#pragma no_pch` can be used to suppress the generation or reuse of precompiled headers for a given source file.

- The option `-Z pch_messages` can be used to instruct the compiler to issue a message confirming the a PCH file is being generated or reused. For example:
  `test.C : CFE1633 creating precompiled header file "test.pch"`
  or
  `test.C : CFE1632 using precompiled header file "test.pch"`

  `-Z no_pch_messages` is the default setting.

5 Optimization

5.1 Overview and general remarks on optimization

The compiler offers a range of optimization possibilities, controlled by means of options, which allow optimization of the program code in such a way that its runtime or (in some cases) its storage space requirement is reduced.

The kernel consists of the optimizations carried out by the compiler backend (see table 1, on page 92). These optimizations are performed on the basis of an intermediate source code (Ucode) which supports optimizations across object modules particularly well. These optimizations are described in detail in the present chapter, starting at section 5.2.

Besides the Ucode optimizations, there are also other optimization options which are based on the C/C++ source code, the ULS intermediate language, and the assembler code (see table 2, page 92). Detailed information on these optimizations can be found in chapter 3, which deals with each optimization option individually.

One of the important concepts that frequently appears in the context of optimization is that of a “basic block”. A basic block represents the largest set of sequential statements without a branch. Each such statement sequence has exactly one entry point and one exit.

	Optimization levels				
Optimizations	`-O`	`-F O2`	`-F O3`	`-F O4`	`-F O5`
Global optimizations	X	X	X	X	X
Loop unrolling controlled by `-F loopunroll,`*n* `-F unrolllimit,`*n*	X	X	X	X	X

Function inlining
controlled by
`-F i,`*file*
`-F no_inlining`
`-F inline_limit,`*n*
`-F sortedges`
`-F space_time,`*n*
		X	X	X	X

Interprocedural register allocation			X		X

Feedback-directed optimizations
controlled by
`-F default_inling`
`-F no_positioning`
`-F no_splitting`
`-F no_br_elimination`
`-F no_feedback_uopt`
`-F default_xbb`
`-F select_ucode,`*n*
`-F profdata,`*“pattern“*
`-F profdir,dvz`
`-F feedback_summary,`*name*
				X	X

Table 1: Ucode-based optimizations and optimization levels

Optimizations	**Option**	**Described on**
Function inlining with C libraries (C frontend)	`-F I` [*name*]	page 56
Elimination of unnecessary code (ULS)	`-F U`	page 58
Alignment of function entries to 2***n* bytes	`-F afep,`*n*	page 55
Addressing global data via the fast gp register	`-K no_pic` `-F G`*n*	page 60 page 55
Profiling (prof, lprof)	`-q p` `-q l`	page 62

Table 2: Other (non-Ucode-based) optimizations with respective options

Optimizations	Option	Described on
Recommended combination of optimizations for R4000 or R10000 systems	`-F X` `-F X4`	page 54

Table 2: Other (non-Ucode-based) optimizations with respective options

General remarks on optimization

It is difficult to predict how successful an optimization attempt will be, as it heavily depends on the specific design of the program or set of programs to be optimized.

A successful optimization will be obtained by alternately testing the effect of individual optimization options, or their most effective combination, on program compilation and program run time.

We recommend measuring the effect of an optimization only when the program runs under standard operating conditions. Otherwise, assumed optimization successes may not be confirmed when the program finally runs under normal working conditions.

The goal of the optimization, i.e. the reduction of program run time, may, however, be in conflict with other goals of the program design:

– Compilation time
  Optimization attempts force the compiler to carry out additional analyses and transformations which take additional compilation time. The more intensive the optimization attempt, the more additional compilation time will be required.

  We recommend only optimizing programs that already compile without any syntax or semantic error.

– Symbolic test
  Since the optimized code does not really allow any cross-reference back to the original source code, it is not possible to subject optimized programs to a symbolic test. When the `-g` option (generate test help information for DBX) is specified, all optimization options except `-F I` and `-F afep,`*n* are ignored (see table 3).

– Size of object
  Some optimizations result in a larger object module, e.g. with function inlining (`-F I`, `-F O`*x*), loop unrolling(`-O`, `-F O`*x*) or function entry alignment (`-F afep`). The increase in size may in some cases decrease program performance (due to inefficient caching). In this case, we recommend finding the best match through trial and error. Special options are available which allow you to control such optimizations individually.

- Portability
  The Position Independent Code (`-K pic`, `-K selpic`), which is required for portable binaries, increases the call overhead and access time to global data. Note that the use of R4000 instructions (`-K mips2`) may preclude execution on R3000 processors which do not support all R4000 instructions.
- Other object properties
  Beside the restriction that optimized programs cannot (except in a few cases) be subjected to a symbolic test, there are further object properties that do not permit optimizations::

Optimization level	**symbolic test** `-g`	**ABI conform** `-K pic`	**dynamic linking** `-d y`	**shareable object** `-G`
`-F I`	+	+	+	+
`-F U`	–	+	+	+
`-F afep,`*n*	+	+	+	+
`-O`	–	+	+	+
`-F O2`	–	+	+	–
`-F O3`	–	–	–	–
`-F O4`	–	+	+	–
`-F O5`	–	–	–	–

Table 3: Optimization levels and possible object properties

5.2 Optimization phases in Ucode-based optimizations

figure 2 shows the different phases the compiler goes through when you carry out a Ucode optimization with the options `-O`, `-F O2`, `-F O3`, `-F O4` or `-F O5` .

In the **uld** phase (Ucode Link), the Ucode of all compilation modules is gathered into one module which is then used to perform the individual optimizations across modules.

The **uopt** phase (Global Optimizer) is used to do global optimizations such as branch elimination, loop unrolling or interprocedural register allocation. If the `-O` option is specified, these optimizations only apply to one module (uld phase is skipped). If the `-F O`*x* options are specified, they are carried out across modules based on the linked Ucode.

The **umerge** phase (Procedure Merge) is used to do function inlining (`-F O`*x*) and interprocedural register allocation (only with `-F O3` and `-F O5`). Additionally, the functions that need to be addressed in the uopt phase are identified. This permits the uopt phase to run more efficiently.

The **uprof** and **upos** phases (Profiling Feedback and Procedure Positioning) are used to carry out feedback-directed optimizations based on previously generated profiling raw data files (`-F O4` and `-F O5`).

In the **ugen** phase (Code Generator), the assembler code is generated from the optimized (`-O`, `-F O`*x*) or unoptimized (`-F ucode`) Ucode.

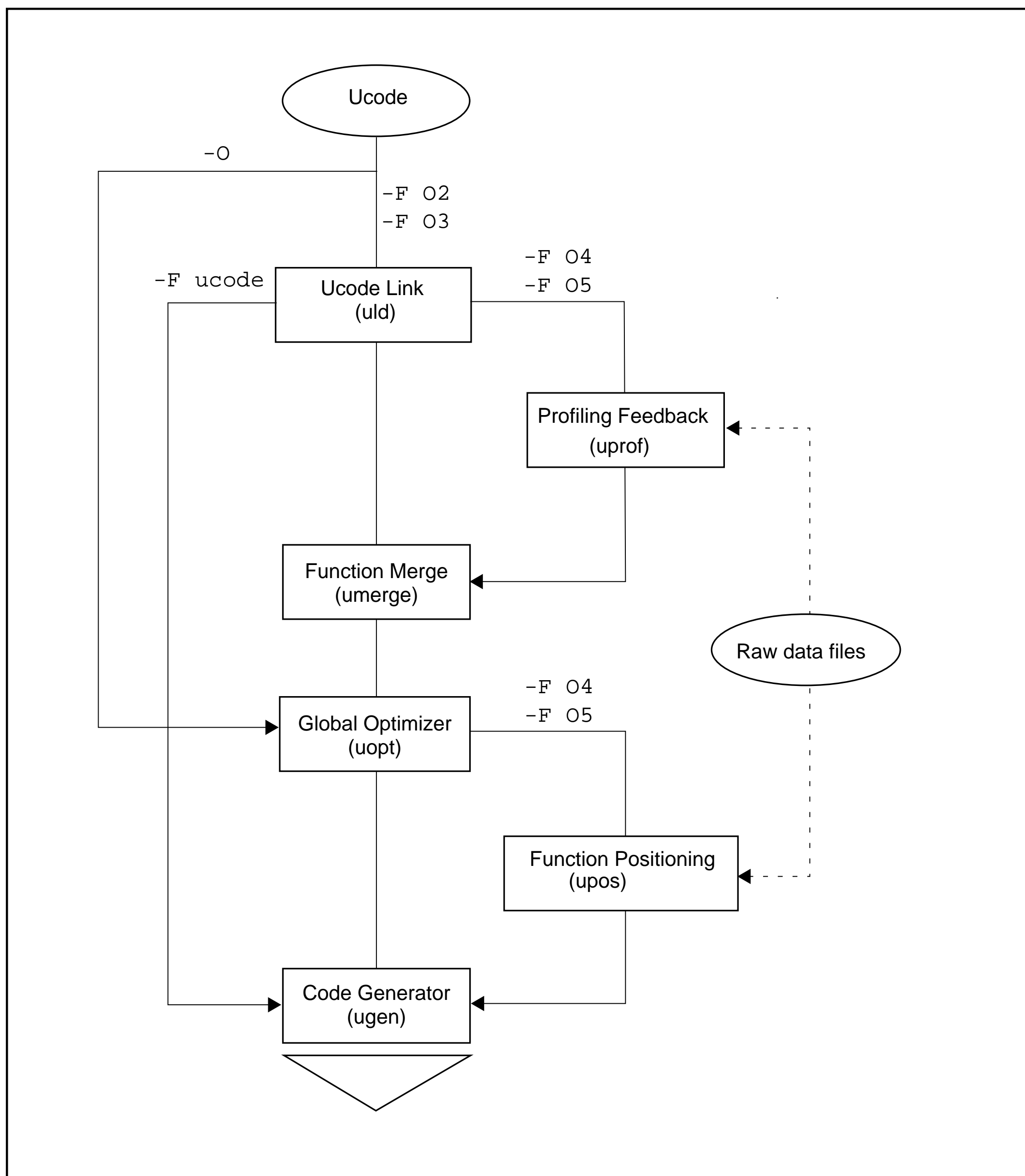


Figure 2: Ucode optimization phases of the compiler

5.3 Ucode and dual object file

The -O, -F O2, -F O3, -F O4 or -F O5 optimization steps are based on Ucode. Ucode is an internally generated intermediate source code (from the ULS intermediate code) which is better suited for optimizations, particularly across object modules and for feedback-directed optimizations.

Optimizations with the -F O*x* options first link - in the uld phase (Ucode link) - the Ucode from the various compilation units to a single unit, which is then used to carry out the optimizations across modules. The ELF object code is not generated and the program is not linked until all optimizations have been completed. In cases where the programs are first compiled separately and linked and optimized in a subsequent step, the unoptimized Ucode generated must be stored in intermediate files (in Ucode files or dual object files, see below).

“Unoptimized Ucode” means that Ucode-based optimizations (see the relevant steps above) are not carried out. The Ucode may, however, already be “optimized” as optimizations based on C/C++ source code and/or on ULS intermediate code may have taken place (see the options -F I “Inlining of C library functions” on page 56 and -F U “Elimination of unneeded code” on page 58).

Creating and processing Ucode files

If source code is compiled using the -c option and the -F O*x* optimization options, each compilation unit creates an object file named *file*.o which contains only unoptimized Ucode. These “Ucode files” can only be processed further using the cc/c89/CC command and again the -F O*x* optimization options. Only then, in the uld phase, is the Ucode collected, the optimization carried out and the optimized object code linked into an executable. If you want to create an unoptimized program from the Ucode files, you must use the option -F ucode instead of -F O*x*.

Creating and processing dual object files

If source code is compiled using the -K dual option together with -c, each compilation unit creates an object file named *file*.o which contains both ELF object code (possibly optimized with the options -O, -F I, -F U) and unoptimized Ucode. These “dual object files” can then be linked like any ELF object and processed further like a Ucode file using the cc/c89/CC command (see above). Dual object files are particularly suited for e.g. feedback-directed optimization which first creates an instrumented program from the unoptimized object code and then the actual optimized program from the Ucode (see also page 103ff).

Examples

Compiler call	Generated object	Further processing
cc -c *file*.c	*file*.o unoptimized ELF code	Linking: cc *file*.o
cc -c -O -F U *file*.c	*file*.o optimized ELF code	cc *file*.o
cc -c -F 0*x* *file*.c	*file*.o Ucode file with unoptimized Ucode	Optimization and linking: cc -F 0*x* *file*.o
cc -c -K dual *file*.c	*file*.o dual object file with ELF code and unoptimized Ucode	Linking of the object code: cc *file*.o
		Optimization and linking of the Ucode: cc -F 0*x* *file*.o

5.4 Global optimizations

In the uopt phase the compiler performs amongst other things, the following global optimizations:

- Elimination of common subexpressions
- Elimination of partial redundancies
- Propagation of constant expressions
- Optimization of pointer references
- General loop optimization (e.g. moving code from within the loop to outside the loop, changing time intensive operations into less time-intensive ones)
- Dead code elimination
- register allocation
- Loop unrolling

If the `-O` option is specified, only these optimizations are carried out and they are limited to the bounds of the individual compilation unit.
If the optimization options `-F O`*x* (*x* = 2, 3, 4 or 5) are specified, they are carried out along with others, such as function inlining or feedback-directed optimizations. In this case, the optimizations are performed across all object modules.

Global optimizations can be supported by appropriate programming (see the programming guidelines on page 101). Furthermore, loop unrolling may be controlled by additional options which will be described further below.

Loop unrolling

Unrolling loops means that the number of cycles through the loop will be reduced by repeating, “unrolling”, the body of the loop one or several times. In every cycle through the loop, control statements must be executed which test for the current number of iterations and then branch accordingly. Reducing the number of iterations of the loop will , therefore, optimize the execution time.

For example, if the body of the loop is repeated twice (unroll factor of 2), the execution time for control statements is halved. As a general rule, an unroll factor of n reduces the execution time for control statements in the loop to $1/n$.
The default setting for the unroll factor is 4.

Unrolling loops also provides new optimization opportunities.The unrolled loop bodies allow for additional optimizations such as propagation of constant expressions and elimination of redundant expressions.

The repetition of code does, however, increase the size of the generated module, which in turn could degrade performance due to inefficient caching. We therefore recommend that the best parameters for loop unrolling be determined by trial and error in some cases. The following options may be used to achieve this:

– With `-F loopunroll,`*n* you can modify the default setting of 4 for the unroll factor. If the factor is set to 0, loop unrolling will be disabled.

– With `-F unrolllimit,`*n* you can determine the maximum number of instructions that loop unrolling may cover. Default setting is *n*=500.

Example for loop unrolling of factor 4

Before unrolling:

```
i = 0;

while(i < 80) {
    a[i] = b[i+1];
    i++;
    }
```

After unrolling:

```
i = 0;

while(i < 80) {
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
    if(!(i < 80)) goto end;
    a[i] = b[i+1];
    i++;
    }
end:
```

With additional optimization opportunities in the body of the loop!

Programming guidelines to support global optimization

Global optimization can be supported by the following programming guidelines:

- Avoid indirect calls such as pointers to functions as arguments.
- Avoid unions that cause overlap between integer and floating-point data types.
- Avoid using global variables, pointers and addresses, where possible. Here a few tips:
  - Declare any variable outside of a function as `static`, unless it is referenced by another source file
  - Implement the data exchange between functions by passing parameters instead of using global variables
  - Use subscripting arrays instead of dereferencing pointers, e.g. `char array[i][j]` instead of `char **array`.
- Avoid functions that take a variable number of arguments (VARARGS).

5.5 Optimization with function inlining

Optimization at the `-F O`*x* (*x* = 2, 3, 4 or 5) level is performed with function inlining of user-defined functions.

Function inlining saves calling the function at execution time because the function code of the called function is copied to the point of call of the calling function. This permits savings on time-consuming code sequences for function calls and return jumps and, therefore, allows considerable savings in execution time. Function inlining also enhances the impact of global optimizations due to the enlarged context.

Function inlining, however, also increases the size of the generated object modules. This must be weighed against the advantages of optimization.

Function inlining can be controlled with the following options:
`-F no_inlining` (see page 57)
`-F i,`*file* (see page 57)
`-F inline_limit,`*n* (see page 57)
`-F sortedges` (see page 58)
`-F space_time,`*n* (see page 58)

Function inlining at `-F O4` and `-F O5` optimization level may be complemented by feedback information (see page 104).

Function inlining should be performed primarily for small functions, as the overhead for function calls and return jumps is quite substantial compared to the actual function code.

Function inlining cannot be performed for functions having the following properties:

- Functions taking any number of variables (see va_... macros in <stdarg.h>)
- Functions with setjmp calls
- Recursive functions

Note on inline functions in C++

Functions defined within classes and functions with the attribute `inline` are C++-specific inline functions. In the C++ modes (only the `CC` command), these C++-specific inline functions are automatically expanded inline, regardless of whether or not optimization is enabled (by setting the `-F O`*x* optimization levels). Apart from the option `-F no_inlining` , none of the other options to individually control inline expansion (see above) have any effect on the handling of inline functions in C++. The option `-F no_inlining` suppresses the inline expansion of C++-specific inline functions as well and also applies to cases where optimization has been turned off.

5.6 Feedback optimization

Feedback optimizations are performed by default when you specify the `-F O4` or `-F O5` option.
The `-F O4` option causes the compiler to additionally perform the same optimizations as those performed at the `-F O2` level (global optimizations and function inlining).
The `-F O5` option causes the compiler to perform additionally the same optimizations as those performed at the `-F O3` level (global optimizations, function inlining and interprocedural register allocation).

Feedback optimizations use profiling information based upon typical program usage determined through test runs of the program which has not yet been optimized. The profiling raw data includes call graph and flow graph information. For each function the call count and how many times the program called each function, is recorded during the profiling run. Flow graph information describes the execution path of the program, noting how often the main path has been left by taking particular branches.

The following optimizations using profiling data are supported:

- Feedback-directed function inlining
- Procedure positioning
- Branch elimination
- Procedure splitting
- Feedback-directed register allocation
- Feedback-directed Pipeline optimization (cross basic block instruction scheduling)

The detail of these feedback-directed optimizations and how they can be controlled through the additional options is described in the next chapter on page 104.

Feedback optimizations require, in principle, the following three steps to implement:

1. First build an unoptimized executable with the `-q f` option, a so-called profiling executable that will generate profiling data when executed. In general, the profiling executable is not generated in one single run but requires the following partial steps:

   - Compiling the source code using the `-c` and `-q f` options. This will produce ELF object code that is instrumented to collect profiling data.
   - Linking the instrumented object files with the cc/c89/CC command, using again the `-q f` option. At this point, additional information is added to the profiling executable that will be relevant at a later processing stage (e.g. the name of the profiling directory).

   See section “Building the profiling executable” on page 106.

2. Secondly, the profiling executable is called one or several times and is fed with the typical usage data. Each time the executable is run, a profiling raw data file is produced as feedback which will later be used to perform the optimizations (see 3.).
   See the section on “Generating the profiling data” on page 109.

3. Finally, the actual optimized program is produced. This requires a new compilation and linkage run with the cc/c89/CC command, using again the `-F O4` or `-F O5` option. The individual compilation phases make use of the raw data files generated in step 2 and perform the appropriate optimizations by evaluating the relevant profiling feedback found.
   See the section on “Producing the optimized program” on page 110.

5.6.1 Feedback optimization features

Feedback optimizations are performed amongst other things, to ensure an optimal usage of the cache. The cache is loaded block by block together with the relevant instructions to be executed. Compressed and efficient supply of the cache blocks with executable code, clearly improves the program´s performance (reduced code thrashing) .

Feedback optimization	Description
Function inlining	Uses profiling information to assess more accurately the benefits and costs of inlining a function (execution time saved versus size of module).
Procedure positioning	Uses profiling data (call graph information) to reorder functions in memory in order to reduce instruction cache conflicts (thrashing). Functions that call each other frequently are arranged in memory so that they do not displace each other.
Branch elimination	Uses profiling data (flow graph information) to reduce the number of branches taken (at machine code level). Branch elimination makes use of the fact that fall-through code (target code after branching) is often already loaded in the same block of the instruction cache and, therefore, makes branching obsolete.
Procedure splitting	Uses profiling data (flow graph information) to move instructions in a function, that are rarely or never executed, to a special area further away from the function body. This works with procedure positioning to improve instruction cache usage.
Register allocation	Uses profiling data for interprocedural register allocation.
Pipeline optimization	Uses profiling data for cross basic block instruction scheduling.

These optimizations can be disabled individually with the following options:

`-F default_inlining`
: Suppresses the use of feedback information during function inlining, i.e. the optimization performed is the same as when the `-F O2` and `-F O3` options are specified.

`-F no_inlining`
: Disables all function inlining.

`-F no_positioning`
: Disables procedure positioning.

`-F no_splitting`
: Disables procedure splitting.

`-F no_br_elimination`
: Disables both branch elimination and procedure splitting.

`-F no_feedback_uopt`
: Suppresses the use of feedback information during register optimization, i.e. the optimization performed is the same as when the `-F O3` option is specified.

`-F default_xbb`
: Suppresses the use of feedback information during cross basic block instruction scheduling (see also `-F noxbb` on page 57).

5.6.2 Building the profiling executable

A profiling executable consists of:

- instrumented object code, i.e. ELF object code with additional information that enables the generation of profiling raw data w.r.t. function calls and graphs, and
- administrative data that will influence the subsequent execution of the executable such as the names of profiling data files and profiling directories.

The instrumented object code is generated when compiling the C/C++ source code. The relevant administrative data is only added to the executable during linking.

Using the `-q f` option, the profiling executable can also be built in a single run combining compiling and linking. In daily practice, however, this remains the exception. Normally, the C sources are compiled separately and the executable is generated in a separate linkage run. The two-step approach for building profiling executables requires that the `-q f` option is set both for compiling and linking.

Preliminary compilation of C/C++ sources

Compiling C/C++ sources with the `-c` and `-q f` options causes the compiler to produce instrumented ELF object code which it writes into an object file named *file*.o. This object file is then used as input source for the subsequent linkage run, during which the profiling executable is generated. When the actual optimized program is finally produced (see page 110), the C/C++ sources must be compiled again.

If the compilation is done using not only `-c` and `-q f`, but also the additional `-K dual` option, the instrumented object code is written into a dual object file named *file*.o. This dual object file can then be used to generate both the profiling executable and the optimized program without having to compile again.

Instrumented object code cannot be generated from assembler code (file.s).

Any additional options such as `-F profdata` and `-F profdir` will be ignored, since such information will be added to the executable only during linking.

Examples

```
$ cd /usr/myself/workdir
$ cc -c -q f file1.c
$ cc -c -q f file2.c
```

The compiler generates instrumented object code and writes it into the object files `file1.o` and `file2.o`.

```
$ cd /usr/myself/workdir
$ cc -c -q f -K dual filex.c
$ cc -c -q f -K dual filey.c
```

The compiler generates instrumented object code and writes it into the dual object files `filex.o` and `filey.o`.

Building the profiling executable

If the `-q f` option is specified without the `-c` option, the optimizer generates a profiling executable. It is named `a.out` or according to the name given with the `-o` option. Input sources may be:

- Object files or dual object files in which the object code has already been instrumented (see page 107).
- C/C++ source files, Ucode files and dual object files in which the object code has not yet been instrumented. In these cases, instrumented object code is produced from the C/C++ source code (case of C source files) or from the Ucode (case of Ucode files or dual files).
- Object files which only hold instrumented code. In this case, the object is linked into a profiling executable, but it is not instrumented for the generation of profiling data.

The compiler generates a profiling directory (if it does not already exist) called `PROF` as subdirectory of the current working directory. This standard profiling directory is also known as *$PWD*/`PROF`.

The compiler stores a copy of the profiling executable called *executable_file*.fb in the profiling directory. This copy will be used for the subsequent feedback-directed optimizations and must not be renamed or deleted. Each time the profiling executable is run, it generates in the profiling directory a new raw data file called *PID.executable_file* by default, *PID* representing the process ID number and *executable_file* the `a.out` or the name determined by the `-o` option.

The compiler will delete any raw data file from the profiling directory which complies to the pattern *PID.executable_file*.

The `-F profdir`, *dir* option allows you to specify the pathname of your own directory, in which you want to store the copy of the profiling program (`.fb` file) or the data files generated from running the profiling executable. The default pathname for the profiling directory is *$PWD*/`PROF` (see above).

With the `-F profdata,'`*pattern*`'` option, the compiler deletes all raw data files generated by the instrumented object and complying with '*pattern*'. The pattern created as '**.name*' causes the compiler to delete the *PID.name* data files instead of the *PID.executable_file* data files.

Example

```
$ cd /usr/myself/workdir
$ cc -q f -o hello file1.o file2.o filex.o filey.o main.c
```

The profiling executable `hello` is generated from the following input sources:
From the object files (`file1.o`, `file2.o`) produced by a former compilation run, from the dual object files (`filex.o`, `filey.o`) with already instrumented object code (see example of page 107) and from the source file `main.c`.
The compiler creates a directory (if it does not already exist) `/usr/myself/workdir/PROF`. If possible, it will delete all files from this site that correspond to the pattern *.hello.
The profiling executable is then stored under the name `hello` in the current working directory `workdir`, a copy of this program is stored as `hello.fb` in the *$PWD*`/PROF` profiling directory.

5.6.3 Generating the profiling data

The profiling executable (see page 106) generated with the `-q f` option is called in the same way as a normal program.

Every time the profiling executable is run, a new data file obeying the name convention *PID.executable_file* is generated, *PID* representing the appropriate process ID number and *executable_file* the name of the profiling executable.These data files are stored in the *$PWD*`/PROF` directory or the profiling directory specified when generating the profiling executable with the `-F profdir` option.

In order to make the right assumptions for the subsequent feedback optimizations, it is necessary to use data for the test runs which reflect typical application usage.

Example

```
$ cd /usr/myself/workdir
$ hello
```

Whenever the profiling executable `hello` is run, a raw data file named *PID*`.hello` (where *PID* = the respective process ID) is stored in the profiling directory `/usr/myself/workdir/PROF`.

5.6.4 Producing the optimized program

The optimized program is produced in a combined compilation and linkage run using the cc/c89/CC command and specifying the `-F O4` or `-F O5` options. Input sources may be C source files, Ucode files and dual object files (see also page 97). Object files containing solely ELF object code can be linked, but no optimizations will be performed for these program modules.

In the case of feedback optimization, the compiler searches by default for raw data files named **.executable_file* in the subdirectory `PROF` of the current working directory. These are the data files previously created when running the profiling executable (see page 109). The profiling information from the individual data files is gathered in a summary data file named *executable_file.summary* which is stored in the *$PWD*/`PROF` profiling directory. The summary data file is then evaluated during the individual optimization phases.
This default processing may be altered with the following options:

`-F profdata,`*'pattern'*
: If this option is specified, the compiler searches for the raw data files corresponding to the naming pattern. The pattern in inverted commas '`*.`*name*' causes the compiler to search for the *PID*`.`*name* data files instead of for *PID.executable_file* files and to produce the summary data file.

`-F profdir,`*dir*
: This option allows you to create a profiling directory named *dir* in which the compiler will search for the raw data files and store the summary data file.
Without this option, searching/storing of the data files will be carried out in the subdirectory `PROF` of the current working directory.

`-F feedback_summary,`*name*
: The compiler does not create a summary data file but evaluates the summary data file specified with the *name* option for feedback optimizations.

Example

```
$ cd /usr/myself/workdir
$ cc -c -F O4 main.c file1.c file2.c
$ cc -o hello -F O4 file1.o file2.o filex.o filey.o main.o
```

The optimized program `hello` is generated from the following input sources:
From the Ucode files (`main.o`, `file1.o`, `file2.o`) produced by a former compilation run and from the dual object files (`filex.o`, `filey.o`) which were used previously to produce the profiling executable (see example on page 107)
The compiler evaluates the data files *.hello in directory `/usr/myself/workdir/PROF`.
They are the data files previously generated by the profiling executable (see example on page109). It creates the summary data file called `hello.summary` from the raw data files in the same profiling directory and evaluates this summary file when performing the feedback optimizations.

6 The link editor

Linking refers to the process in which the **reference** to a symbol in one module of your program is connected with its **definition** in another. Whichever linking model you choose, static or dynamic, the link editor will search each module of your program, including any libraries you have used, for definitions of undefined external symbols in the other modules. If it does not find a definition for a referenced symbol, the link editor will report an error and fail to create an executable program. The principal difference between static and dynamic linking lies in what happens after the resolution of undefined external symbols:

- Under static linking, copies of the archive library object files that satisfy unresolved external references in your program are incorporated in your executable at link time. External references in your program are connected with their definitions - assigned addresses in memory - when the executable is created.
- Under dynamic linking, the contents of a shareable object are mapped into the virtual address space of your process at runtime. External references in your program are not connected with their definitions until the program is executed.

A dynamically linked executable normally will use less disk space than a statically linked executable because shareable object code is not copied into the executable object file at link time. For the same reason, shareable object code can be changed without having to relink executables that depend on it. Even if the shareable C library were enhanced in the future, you would not have to relink programs that depended on it as long as the enhancements were compatible with your code (see the section “Checking for runtime compatibility”). The dynamic link editor would simply use the definitions in the new version of the library to resolve external references in your executables at runtime.

Static linking

There are certain cases in which dynamic linking may not be preferable to static linking because the former mode might require more space.

- With programs which call only a few small library functions, the bookkeeping information to be used by the dynamic linker may take up more space in your executable than the code for those functions. You can use the `size` command (see “Programmer’s Reference Manual”), to determine the difference in the sizes of the object files.
- Static linking is also preferable where using a shareable object may add to the memory requirements of a process. Although a shareable object’s text is shared by all processes that use it, its data typically is not (at least its writable data; see page 118). Every process that uses a shareable object usually gets a private copy of its entire data segment, regardless of how much of the data is needed. If an application uses only a small portion of a shareable library’s text and data, executing the application might require more memory with a shareable object than without one. It would be unwise, for example, to use the standard C shareable object library to access only `strcmp()`. Although sharing `strcmp()` saves space on your disk and memory on the system, the memory cost to your process of having a private copy of the C library’s data segment would make the statically linked version the more appropriate choice.

If you plan a shareable library for commercial use, you ought always to provide a compatible static version as well. You can use the same source files to build the static and dynamic versions of a library.

Dynamic linking

Each process that uses a shareable object references a single copy of the object code in memory. That means that when other processes on your system call a function in a shareable object library, the entire contents of that library are mapped into the virtual address space of these processes as well. If they have called the same function as you, external references to the function in the respective programs will, in all likelihood, be assigned different virtual addresses. That is, because the function may be loaded at a different virtual address for each process that uses it, the system cannot calculate absolute addresses in memory until runtime.

The memory management scheme underlying dynamic linking shares memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at runtime. If a process writes to a shared page in the course of relocating a reference to a shareable object, it gets a private copy of that page and loses the benefits of code sharing. Other users of the page remain unaffected.

The link editor will not report an error for multiple definitions of a function or a same-sized data object when each such definition resides within a different shareable object or within a dynamically linked executable and different shareable objects. The dynamic linker will use the definition in whichever object occurs first on the command line. You can, however, specify `-B symbolic` when you create a shareable object with

```
$ cc -G -B symbolic -o libfoo.so function1.c function2.c function3.c
```

to ensure that the dynamic linker will use the shareable object's definition of one of its own symbols, rather than a definition of the same symbol in an executable or another library.

Dynamic linking offers the following advantages:

- Library code is not copied into the executables that use it. Apart from the exceptions above, the executables therefore require less disk space.
- Because library code is shared at runtime, the memory needs of systems at runtime are usually reduced.
- Because symbol resolution is put off until runtime, shareable objects can be updated without having to relink applications that depend on them.
- As long as its pathname is not hard-coded in an executable, a shareable object can be moved to a different directory without having to relink an application.

A programming interface is provided for the dynamic linking mechanism to attach a shareable object to the address space of your process during execution, look up the address of a function in the library, call that function, and then detach the library when it is no longer needed. The functions for this are stored in the library `libdl.so`. Its contents are described in the “Programmer's Reference Manual”.

6.1 Default settings and standard libraries

If no options are specified in the cc/c89/CC call to suppress the linkage run (see the section “Options to select compilation phases” on page 30), object files are created for the individual source files and then linked into an executable program. This executable is given the name a.out by default or the name specified with the -o option (see page 27).

Dynamic linking (-d y; see page 66) is set by default. In other words, if the -l *x* option is specified, the link editor will normally look for the shared version of the library (lib*x*.so) first. If no shared version is available, the static version of the library (lib*x*.a) is used.

If a lib*x*.a or lib*x*.so library is not present in a directory that is automatically searched by the link editor (see “Standard directories, which are searched last” on page 38), the appropriate directory must be specified with the -L option (see page 69). For example, the current directory can be searched by specifying a dot (.) as the directory name: -L .

Linking C and C++ standard libraries

Some C and C++ standard libraries are linked automatically. This means that the corresponding -l options are automatically set for these libraries internally (see also the option -K link_stdlibs on page 67). Other C and C++ standard libraries, by contrast, must be explicitly specified with the -l option. For the sake of simplicity, only the names of the static versions of these libraries (suffix .a) are listed below. Each of these libraries, except for the C math library, is also available as a shared library (suffix .so).

- For the cc and c89 commands
  - Standard C library (libc.a): automatic
  - C math library: -l m
- For the CC command in Cfront C++ mode (-X d)
  - Cfront iostream library (libC.a): automatic
  - Cfront complex library: -l complex
  - Standard C++ library: -l CFstd
  - Tools.h++: -l CFtools
  - C libraries: see cc and c89

  To ensure that the Cfront libraries are linked correctly, the language mode option -X d must be specified for compilation as well as linkage:

```
CC -X d -c x.C y.C        # Compilation
CC -X d -l m x.o y.o      # Linkage
```

- For the `CC` command in the ANSI C++ modes (`-X w`, `-X e`)
  - Standard C++ library, including the Cfront iostream (`libCstd.a`) : automatic
  - Tools.h++: `-l RWtools`
  - C libraries: see `cc` and `c89`

6.2 Creating your own libraries

This section describes how to create statically and dynamically linked libraries.

6.2.1 Creating a statically linked library

The following commands:

```
$ c89 -c function1.c function2.c function3.c
```

```
$ ar -r libfoo.a function1.o function2.o function3.o
```

will create the statically linked ISO/ANSI-conformant library, `libfoo.a`, that consists of the named object files (see also “Programmer’s Reference Manual”). When you use the `-l` option to link your program with `libfoo.a`:

```
$ c89 -L dir file1.c file2.c file3.c -l foo
```

the link editor will incorporate in your executable only those object files which contain a function you have called in your program. Note that because the dynamic linking default was not turned off with the `-dn` option, the link editor will use the dynamically linked version `libc.so` as well as the statically linked version `libfoo.a`.
The option `-L` *dir* is used to point the link editor to the directory in which your library is stored.

6.2.2 Creating a dynamically linked library

You create a dynamically linked library by specifying the `-G` option to the link editor. However, this is only possible in PIC code (`-K pic`).

Example 1

```
$ cc -G -o libfoo.so function1.o function2.o
```

This command will create the shareable object `libfoo.so`, consisting of the object code for the functions contained in the named files. Without specifying the -o option, the shareable object would be written into `a.out`.
Use the `-l` option to link your program with `libfoo.so` in the following way:

```
$ cc -o program -L . file.c -l foo
```

The option `-L .` causes the link editor to search for the `libfoo.so` library in the working directory.
The link editor will record in your executable `program` the name of the shareable object `libfoo.so`. It is the dynamic link editor (`ldd`) that then loads at runtime the modules of the shareable object `libfoo.so`.

Before the program `program` is called, the directory which holds library `libfoo.so` must be indicated to the dynamic link editor. This is achieved by means of the environment variable `LD_LIBARAY_PATH`:

```
$ LD_LIBRARY_PATH=. ; export LD_LIBARY_PATH
$ program
```

Example 2

```
$ cc -G -h libfoo.so -o my_object.so function1.o function2.o
```

This command generates the shareable object `my_object.so`. The link editor records, as a reference, the library name `libfoo.so` specified with the `-h` option in the dynamic portion of the shareable object at runtime. The same name will also be recorded in the executable at compilation time. At runtime, the dynamic link editor (`ldd`) will link the modules of the shareable object `libfoo.so`.

```
$ cc -o program file.c my_object.so
```

This command generates an executable `program` which resolves the references from library `my_object.so` and links the modules from library `libfoo.so` at runtime.

```
$ ldd program
$ cp my_object.so libfoo.so
$ program
```

Command `ldd` can be used to check which shareable libraries will be linked at runtime.

6.3 Enhancing performance

There are two ways to enhance shareable library performance: minimizing the library's data segment and minimizing page faults.

6.3.1 Minimizing the data segment

Only the text segment of a shareable object is shared by all processes that use it; its data segment typically is not. Every process that uses a shareable object usually gets a private memory copy of its entire data segment, regardless of how much of the data is needed. You can cut down the size of the data segment in a number of ways:

- Try to use (stack) variables of the memory class `auto` instead of static storage space.
- Use functional interfaces rather than global variables. The code will then be easier to maintain. Often there is no need for global variables.

  Example of message output:

  First, the compilation system reserves memory for the table in the address space of each executable that uses a global table or shareable library, even though it does not know yet where the table will be loaded. After the table is loaded, the dynamic linker copies it into the space that has been set aside. Each process that uses the table, then, gets a private copy of the library's data segment, including the table, and an additional copy of the table in its own data segment. Moreover, each process pays a performance penalty for the overhead of copying the table at runtime. Finally, because the space for the table is allocated when the executable is built, the application will not have enough room to hold any new messages you might want to add in the future. A functional interface enables these difficulties to be overcome.

  `strerror()` might be implemented as follows:

```
static const char *msg[] = {
        "Error 0",
        "Not owner",
        "No such file or directory",
        ...
};
char *
strerror(int err){
        if (err < 0 ||    err >= sizeof(msg)/sizeof(msg[0]))
                return 0;
        return (char *)msg[err];
}
```

Using the type attribute `const` means that the table is defined as read-only. In contrast to data that can be modified, data that is defined as read-only is stored in the text segment (see also `–K roconst`). Because no application copy exists in this case, the dynamic linker does not waste time moving the table. New messages can be added, because only the library knows how many messages exist.

In a similar way, you should try to allocate buffers dynamically, not statically. That will save memory because only the processes that need the buffers will get them. It will also allow the size of the buffers to change from one release of the library to the next without affecting compatibility.

Example:

```
char *buffer()
{
    static char *buf = 0;
    if (buf == 0)
    {
            if ((buf = malloc(BUFSIZE)) == 0)
                    return 0;
    }
    ...
    return buf;
}
```

- You can minimize the data segment by making the library self-contained. If a shareable object imports definitions from another shareable object, each process that uses that object will get a private copy not only of its data segment, but of the data segment of the shareable object from which the definitions were imported.
- Exclude functions that use large amounts of global data. If an infrequently used routine defines a great deal of static data, it probably does not belong in a shareable library.

6.3.2 Minimizing page faults

Although processes that use shareable libraries will not write to shared pages, they still may incur page faults. To the extent they do, their performance will degrade. You can minimize page faults in the following ways:

- Organize to improve locality of reference. First, exclude infrequently used routines on which the library itself does not depend. Traditional `a.out` files contain all the code they need at runtime. So if a process calls a function, it may already be in memory because of its proximity to other text in the process. If the function is in a shareable library, however, the surrounding library code may be unrelated to the calling process. Only rarely, for example, will any single executable use everything in the shareable C library. If a shareable library has unrelated functions, there may be more page faults. Functions used by only a few files do not save much disk space by being in a shareable library, and can degrade performance.
- Try to group dynamically related functions. If every call to `funcA()` generates calls to `funcB()` and `funcC()`, try to put them in the same page. The command `cflow`, (see “Programmer’s Reference Manual”), generates this kind of static dependency information. Combine it with profiling to see what functions actually are called.
- Align the functions on page boundaries. Try to arrange the shareable library’s object files so that frequently used functions do not unnecessarily cross page boundaries. First, determine where the page boundaries fall. You can use the `nm` command, (see “Programmer’s Reference Manual”) to determine how symbol addresses relate to page boundaries. Then place the less frequently called functions between the chunks. Because the code between pages is referenced less frequently than the page contents, the probability of a page fault is decreased. You can put frequently used, unrelated functions together because they will probably be called frequently enough to keep the pages in memory.

6.4 Specifying directories

In the previous section we created the archive library `libfoo.a` and the shareable object `libfoo.so`.
Both these libraries are stored in the directory `/home/mylibs`. You create your executable in a different directory.

If you want to link your program with either of these libraries, you must point the link editor to the `/home/mylibs` directory by specifying its pathname with the `-L` option:

```
$ cc -L /home/mylibs file1.c file2.c file3.c -l foo
```

The `-L` option directs the link editor to search for the libraries named with `-l` first in the specified directory, then in the standard file directories. As dynamic linking is turned on by default, the command line above will cause the link editor to search for the library `libfoo.so` in the directory `/home/mylibs`. The link editor will likewise use `libc.so` and not `libc.a`.
Note that you must specify the name of the current directory if it is to be searched by the link editor for libraries. You can use a period (`.`) to represent the current directory.

To direct the link editor to use the statically linked version `libfoo.a`, you can turn off the dynamic linking default:

```
$ cc -d n -L / home/mylibs file1.c file2.c file3.c -l foo
```

If the `-d n` option is specified, the link editor will not accept shareable objects as input. In this example it will search `libfoo.a` rather than `libfoo.so`, and `libc.a` rather than `libc.so`.

To link your program statically with `libfoo.a` and dynamically with `libc.so`, you can do either of two things.

- The first possibility is to move `libfoo.a` to a different directory - `/home/archives`, for example - and then to specify `/home/archives` before the directory `/home/mylibs` by using the `-L` option:

  ```
  $ cc -L /home/archives -L /home/mylibs file1.c file2.c file3.c -l foo
  ```

  Since the link editor encounters the `/home/archives` directory before it encounters the `/home/mylibs` directory, it will use `libfoo.a` rather than `libfoo.so`, since `libfoo.a` is found first.

- A better alternative might be to leave `libfoo.a` in the directory `/home/mylibs` and use the `-B static` and `-B dynamic` options to turn dynamic linking off and on. The following command will link your program statically with `libfoo.a` and dynamically with `libc.so`:

  ```
  $ cc -L /home/mylibs file1.c file2.c file3.c -B static -l foo -B dynamic
  ```

You can use the -B option as a toggle any number of times on any single cc/c89/CC command line. Beginning with the -B static option, the link editor will not accept a shareable object as input until the next -B dynamic or the end of the command line.

```
$ cc -L/home/mylibs file1.c file2.c -Bstatic -lfoo \
  file3.c -B dynamic -l shareableob
```

That command will direct the link editor to search for libraries as follows:

1. libfoo.a, to satisfy unresolved external references in file1.c and file2.c.
2. libshareableob.so, to satisfy unresolved external references in all three files and in libfoo.a.
3. libc.so, to satisfy unresolved external references in all three files and in the two preceding libraries, libfoo.a and libshareableob.so.

Files, including libraries, are searched for definitions in the order in which they are listed on the cc/c89 command line. The standard C library is searched last.

You can add to the list of directories to be searched by the link editor by using the environment variable LD_LIBRARY_PATH.
LD_LIBRARY_PATH must be specified as a contiguous list of colon-separated directory names; an optional second list may be entered, separated from the first by a semicolon:

```
$ LD_LIBRARY_PATH=dir:dir;dir:dir
$ export LD_LIBRARY_PATH
```

The directories specified in the first list before the semicolon are searched first, in order, before the directories specified with -L; the directories specified after the semicolon are searched thereafter. Finally, the standard file directories are searched. Note that you can specify all directories with LD_LIBRARY_PATH and avoid -L altogether. In that case the link editor will search for libraries named with -l first in the directories specified before the semicolon, next in the directories specified after the semicolon, and finally in the standard file directory. You should use absolute pathnames when you set this environment variable.

Note that LD_LIBRARY_PATH is also used by the dynamic linker. If LD_LIBRARY_PATH exists in your environment, the dynamic linker will search the directories named in it for shareable objects to be linked with your program at execution. Note however while using LD_LIBRARY_PATH that any directories that are specified are valid for both link editors.

Specifying directories for the dynamic linker

As mentioned earlier, when you use dynamically linked libraries that are not located in the standard file directories, you must point the dynamic linker to the respective directories. The environment variable `LD_RUN_PATH` lets you do that at link time. To set `LD_RUN_PATH`, list the absolute pathnames of the directories you want searched in the order you want them searched and separate the pathnames with a colon. Since we are only concerned with the directory `/home/mylibs` here, the following will suffice:

```
$ LD_RUN_PATH=/home/mylibs
$ export LD_RUN_PATH
```

Now the command

```
$ cc -o prog -L /home/mylibs file1.c file2.c file3.c -l foo
```

will direct the dynamic linker to search for `libfoo.so` in `/home/mylibs` when the program is executed.

The dynamic linker searches the directories you have assigned to `LD_RUN_PATH` followed by the standard file directory. All executable versions of libraries supplied by the compilation system are kept in `/usr/lib`.
You may specify the directories at the time of execution by using the environment variable `LD_LIBRARY_PATH`, which is set like the environment variable `LD_RUN_PATH`. The set directories are searched before the standard directories.

Suppose you have moved `libfoo.so` to `/home/shareableobs` and would therefore have to link your program again if you use `LD_RUN_PATH` to indicate the new directory. You can, however, assign the new directory to `LD_LIBRARY_PATH`, as follows:

```
$ LD_LIBRARY_PATH=/home/shareableobs
$ export LD_LIBRARY_PATH
```

Now when you execute your program

```
$ prog
```

the dynamic linker will search for `libfoo.so` first in `/home/mylibs` and, not finding it there, in `/home/shareableobs`.
The directories assigned to `LD_RUN_PATH` are searched before those assigned to `LD_LIBRARY_PATH`. The important point is that because the pathname of `libfoo.so` is not hard-coded in `prog`, you can direct the dynamic linker to search a different directory when you execute your program. In other words, you can move a shareable object without having to relink your application.

You can set `LD_LIBRARY_PATH` without first having set `LD_RUN_PATH`. When `LD_RUN_PATH` is used for an application, the dynamic linker will search the specified directories every time the application is executed unless the application has been relinked in a different environment. By contrast, you can use `LD_LIBRARY_PATH` to assign different directories each time you execute the application.

Note that when linking a set-user or set-group ID program, the dynamic linker will ignore any directories specified by LD_LIBRARY_PATH that are not ’trusted.’ Trusted directories are built into the dynamic linker and cannot be modified by the application. Currently, the only trusted directory is /usr/lib.

6.5 Checking for runtime compatibility

When using an updated version of a shareable object you must check that the dynamically linked executable is still compatible with the new version of the library. The program has already been compiled with the previous version. The new version does not have to be linked again! The dynamic linker will simply use the definitions in the new version of the shareable object to satisfy unresolved external references in the executable.

You now need to check that symbols which were defined by the previous version are still defined by the new version

There are two ways you can check for runtime compatibility:

- The first possibility is to use the command `ldd` (see “Programmer’s Reference Manual”). This directs the dynamic linker to print the pathnames of the shareable objects on which your program depends:

  `$ ldd` *prog*

  When you specify the `-d` option to `ldd`, the dynamic linker prints a diagnostic message for each unresolved data reference it would encounter if *prog* were executed. When you specify the `-r` option, it reports each unresolved data or function reference.

- The other possibility is to set the environment variable `LD_BIND_NOW` when you execute your program. The dynamic linker resolves data references immediately at runtime so that unresolved data references are reported prior to execution. By contrast, it normally delays resolving function references until a function is invoked for the first time. This means that the lack of a definition for a function will not be apparent until the function is invoked. You can avoid this by setting the environment variable `LD_BIND_NOW` before you execute your program. The variable `LD_BIND_NOW` is set as soon as you assign any value to it, e.g. the value `1`:

  ```
  $ LD_BIND_NOW=1
  $ export LD_BIND_NOW
  ```

  By doing this, you direct the dynamic linker to resolve all references immediately and can thus ensure before execution that the functions invoked by your process are actually defined.

6.6 Linking library versions

There may be instances in which you need to release a library version that is incompatible with its predecessor. On the one hand, you will want to maintain the older version for dynamically linked executables that depend on it. On the other hand, you will want newly created executables to be linked with the updated version. Moreover, you will probably want both versions to be stored in the same directory. In this situation, you could give the new release a different name, rewrite your documentation, and so forth. A better alternative would be to plan for the contingency in the very first instance by using the following sequence of commands when you create the original version of the shareable object:

```
$ cc -G -h libfoo.1 -o libfoo.1 function1.c function2.c function3.c
$ ln libfoo.1 libfoo.so
```

If PIC code is not the default on your system, the `-k pic` option must be added to the `-G` option in the following examples. In the first command, the `-h` option stores the name given to it, `libfoo.1`, in the shareable object itself. You then use the `ln` command to create a link between the name `libfoo.1` and the name `libfoo.so`. The latter is the name the link editor will look for when it is called via `cc` as follows:

```
$ cc -L dir file1.c file2.c file3.c -l foo
```

The link editor searches for the library `libfoo.so`. However, it will record in the executable the name you gave to `-h`, `libfoo.1`. That means that when you release a subsequent, incompatible version of the library, `libfoo.2`, executables that depend on `libfoo.1` will continue to be linked with the older version, as the dynamic linker uses the name that is stored in the executable.

You use the same sequence of commands when you create `libfoo.2`:

```
$ cc -G -h libfoo.2 -o libfoo.2 function1.c function2.c function4.c
$ ln libfoo.2 libfoo.so
```

With the command:

```
$ cc -L dir file1.c file2.c file3.c -l foo
```

the name `libfoo.2` will be stored in their executables, and their programs will be linked with the new library version at runtime.

7 C language support of the compiler

When the compiler is called with the `cc` or `c89` commands, it operates as a C compiler and optionally supports the C language scope as defined by Kernighan & Ritchie as well as the ANSI/ISO standard (including the ISO C Amendment 1). The default mode for the `cc` command is extended ANSI C; the default for the `c89` command is strict ANSI C (see also the section “Options to select the language mode” on page 34).

The Kernighan & Ritchie definition is documented in:

“The C Programing Language”, B.W. Kernighan and D.M. Ritchie, 1977, Prentice-Hall

The ANSI/ISO definition is documented in:

"The C Programing Language 2nd edition - ANSI C”, B.W. Kernighan and D.M. Ritchie, Prentice-Hall, ISBN 0-13-110370-9

“American National Standard for Information Systems - Programming Language C”, Doc.No. X3J11/90-013, February 14, 1990 and
“International Standard ISO/IEC 9899 : 1990, Programming languages - C”

The ISO C Amendment is documented in:

“International Standard ISO/IEC 9899 : 1990, Programming languages - C / Amendment 1 : 1994”

The following sections, which are intended as a supplement to the vendor-independent literature listed above, describe the implementation and the machine-specific characteristics of this compiler and the various extensions to the standard C language definitions above.

Section 7.1 compares the C language modes of the compiler and points out the most important differences between them.
Section 7.2 describes implementation-defined behavior based on the ANSI/ISO standard, while focusing on aspects that have not been discussed elsewhere in this User Guide or in the “Programmer’s Reference Manual”.
Section 7.3 describes the C language extensions to the definition in the ANSI/ISO standard.
Section 7.4 describes which preprocessor directives are available in addition to those defined in the ANSI/ISO standard and also explains the implementation-specific `#pragma` directive.

7.1 Overview of the C language modes

In accordance with the different language standards defined for C, the compiler supports three C compilation modes:

K&R C mode (option -X t)
The compiler accepts C code based on the language definition by Kernighan & Ritchie as well as some ANSI-specific extensions.

Extended ANSI C mode (option -X a) or strict ANSI C mode (option -X c)
The compiler accepts C code based on the ANSI/ISO definition.

The following table contains an overview of the language elements defined in the ANSI/ISO C standard and indicates which of those elements are supported in the K&R and ANSI C modes.

Key to the entries in the table:

X Fully supported

XE Extension to ANSI/ISO C that is fully supported in strict ANSI C mode, but results in a warning

o Supported syntactically, but not semantically

– Not supported

[1)] to [11)] Notes at the end of the table

C language elements	Language definition		Compilation mode		
	ANSI/ISO	**K&R**	**extended ANSI**	**strict ANSI**	**K&R**
Lexical elements					
Multibyte characters	X	–	X	X	X
Trigraph sequences	X	–	X	X	X
`??=  #`					
`??(  [`					
`??/  \`					
`??)  ]`					
`??'  ^`					
`??<  {`					
`??!  |`					
`??>  }`					
`??-  ~`					
Digraph sequences 1)	X	–	X	X	X
`<:  [`					
`:>  ]`					
`<%  {`					
`%>  }`					
`%:  #`					
`%:%:  ##`					
Escape sequences					
`\a`	X	–	X	X	–
`\b`	X	X	X	X	X
`\f`	X	X	X	X	X
`\n`	X	X	X	X	X
`\r`	X	X	X	X	X
`\t`	X	X	X	X	X
`\v`	X	–	X	X	X
`\'`	X	X	X	X	X
`\"`	X	–	X	X	X
`\?`	X	–	X	X	X
`\\`	X	X	X	X	X
`\` *octdigits*	X	X	X	X	X
`\x` *hexdigits*	X	–	X	X	X
Lengths of identifiers					
internal	31	8	all characters are significant		
external	6	<8			
Keywords 2)					

C language elements	Language definition		Compilation mode		
	ANSI/ISO	K&R	extended ANSI	strict ANSI	K&R
Constants					
integer	X	X	X	X	X
float	X	X	X	X	X
character	X	X	X	X	X
L' character'	X	–	X	X	X
string	X	X	X	X	X
L"string"	X	–	X	X	X
enum	X	X	X	X	X
Suffixes					
integer L,l	X	X	X	X	X
integer U, u	X	–	X	X	X
integer LL,ll 3)	–	–	X	XE	X
float F, f	X	–	X	X	X
float L,l	X	–	X	X	X
Data type declarations					
Type specifiers					
void 4)	X	–	X	X	X
void * 4)	X	–	X	X	X
char	X	X	X	X	X
short	X	X	X	X	X
int	X	X	X	X	X
long	X	X	X	X	X
long long 5)	–	–	X	XE	X
float	X	X	X	X	X
double	X	X	X	X	X
long double	X	–	X	X	X
signed	X	–	X	X	X
unsigned	X	X	X	X	X
array [ ]	X	X	X	X	X
structure 6)	X	X	X	X	X
union 6)	X	X	X	X	X
(*)	X	X	X	X	X
enum	X	X	X	X	X
()	X	X	X	X	X
Type qualifiers					
const	X	–	X	X	o
volatile	X	–	X	X	o
Initialization					
auto aggregate	X	–	X	X	X

C language elements	**Language definition**		**Compilation mode**		
	ANSI/ISO	**K&R**	**extended ANSI**	**strict ANSI**	**K&R**
Storage classes					
typedef	X	X	X	X	X
extern	X	X	X	X	X
static	X	X	X	X	X
auto	X	X	X	X	X
register	X	X	X	X	X
Bitfield types					
int	X	X	X	X	X
signed int	X	X	X	X	X
all integral	–	–	X	XE	X
Conversion rules [7)]					
value preserving	X	–	X	X	–
sign preserving	–	X	–	–	X
Functions					
Definition “old” [8)]	X	X	X	X	X
Definition “new”	X	–	X	X	X
Protoyping [9)]	X	–	X	X	o
Parameter type matching [10)]	X	–	X	X	–
Preprocessor directives					
# (stringizing)	X	–	X	X	X
## (token pasting)	X	–	X	X	X
#assert / #unassert	–	–	X	XE	X
#define	X	X	X	X	X
defined	X	–	X	X	X
#elif	X	–	X	X	X
#else	X	X	X	X	X
#endif	X	X	X	X	X
#error	X	–	X	X	X
#include	X	X	X	X	X
#if	X	X	X	X	X
#ifdef	X	X	X	X	X
#ifndef	X	X	X	X	X
#ident	–	–	X	XE	X

C language elements	Language definition		Compilation mode		
	ANSI/ISO	K&R	extended ANSI	strict ANSI	K&R
#line	X	X	X	X	X
#line (old style)	–	X	X	XE	X
#pragma	X	–	X	X	X
#undef	X	X	X	X	X
# (null directive)	X	–	X	X	X
Predefined macro names					
`__LINE__`					
`__FILE__`	X	–	X	X	X
`__DATE__`	X	–	X	X	X
`__TIME__`	X	–	X	X	X
`__STDC__`	X	–	X	X	X
`__STDC_VERSION__` 11)	X	–	X	X	X
	X	–	X	X	X

Notes

[1] Digraph sequences

Digraph sequences are defined in the ISO C Amendment 1 and are recognized in the C compilation modes only if the option `-K alternative_tokens` is set.

[2] Reserved keywords

asm	continue	extern	long	static	void
auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	

The `asm` keyword is an extension to ANSI/ISO C. It is not reserved in the strict ANSI C mode (but see also `__asm`, on page 147).

[3] Suffixes LL, ll

These suffixes are an extension to ANSI/ISO C and identify integer constants of type `long long` (see page 146).

[4] void

The type void signifies an empty set of values. It can be used in the following three ways:

1. Result type of functions which do not return a value.
2. A pointer to void points to an object of any data type.
3. The number of parameters and the data types of the parameters can be specified in a function declaration (see Prototyping). If void is used instead of the parameter list then no parameters are defined.

All three possibilities are also supported in K&R mode.

[5] long long

This type is an extension to ANSI/ISO C (see page 146).
`long long` is also supported in K&R mode.

6) structure, union

In contrast to the K&R definition, structures and unions may be mutually assigned (if of the same type), passed to functions as parameters, and returned as exit values of functions.
These options are also supported in K&R mode.

7) Implicit arithmetic conversions

One important difference between ANSI and K&R lies in the area of implicit arithmetic conversions.

In K&R mode, operands of an expression are converted using the “unsigned-preserving” rule, i.e. extending an operand of type `unsigned char` or `unsigned short` produces a result of type `unsigned int`. If `unsigned` types appear in an expression together with other types, the result is always `unsigned`.

In ANSI mode, by contrast, the “value-preserving” rule applies, i.e. the result type depends on the size of the operand type. Extending an operand of type `unsigned char` or `unsigned short` thus produces a result of type `int` if `int` is large enough to represent all values of the smaller type. Otherwise, the result is an `unsigned int`.

Due to this difference, the results of arithmetic expressions could differ in some cases and thus lead to erratic program behavior. This must be taken into account when moving from K&R C to ANSI C.

8) Definition of functions

In contrast to K&R, ANSI has introduced a new syntax for the definition of formal function parameters, but also allows the “old-style” (K&R) syntax.
Both definition types are also supported in K&R mode.

9) Prototyping

In contrast to K&R, ANSI defines function prototypes. These are function declarations in which the number and types of individual parameters are also specified. This enables the compiler to compare the types of current parameters with those of formal parameters in the declaration and to adapt them to the formal parameters as required.

Prototype declarations are syntactically allowed in K&R mode, but have no semantic significance.

[10] **Parameter type matching**

The advantage of prototyping is that the parameters specified in the function declaration are not subject to standard conversion rules. A parameter that is declared there as `float` will also be passed as `float`, without first being converted to `double`. If K&R and ANSI objects are to be combined, floating-point parameters should always be declared `double`.

The automatic matching of parameter types is only supported in ANSI modes.

[11] `__STDC_VERSION__`

This preprocessor macro is defined in the ISO C Amendment 1 (see “Predefined preprocessor names” on page 78ff).

7.2 Implementation-defined behavior based on the ANSI/ISO C standard

Interactive device

An interactive device is a device (usually a terminal) that is used by a person or any device that responds like a human being.

Hosted/freestanding environment

CDS++ supports only a networked (hosted) environment in which an operating system (Reliant UNIX) is running. It is not designed to run in a freestanding environment.

Identifiers

Internal and external names can have any length, and all characters are significant. A distinction between uppercase and lowercase letters is made even for external names. In accordance with ANSI/ISO C, the following characters are allowed when constructing names: the digits 0 to 9, the uppercase letters A to Z, the lowercase letters a to z, and the underscore _. Multibyte characters are not supported in identifiers.

As an extension to ANSI/ISO C (controllable by options), the “dollar” character $ and the “at” character @ are also allowed in names (see page 146).

main function

The compiler allows the return types `int` and `void` for the `main` function.

Two formal parameters are provided for the `main` function to allow arguments to be passed to a program in a call:

```
int main(int argc, char *argv[])
```

The names for these parameters (*argc*, *argv*) may be selected arbitrarily, but these are the ones conventionally used in UNIX.

The first parameter *argc* shows the number of passed arguments. Since the first argument, argv[0] (see below), is conventionally the program name, the number of arguments is at least 1.

The second parameter *argv* is a pointer to an array of strings. It holds the program name (in argv[0]) and all arguments entered in the program call in the form of strings terminated with the null byte (\0).

As an extension to ANSI/ISO C, it is also possible to declare a third parameter `char *`*envp*`[]` for the `main` function (see page 146).

Example 1: Passing file names via argv[1]

The following example shows how you can supply your program with a file name. The call:

```
$ prog filename
```

causes `prog` to open the specified file.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
      FILE *fin;
      int ch;
      switch (argc)
      {
      case 2:
            if ((fin = fopen(argv[1], "r")) == NULL)
            {
                /* first string (argv[0]) is the program name. */
                /* second string (argv[1]) is the name of the file,  */
                /* not be opened. */

                (void)fprintf(stderr, "%s: Cannot open input file %s\n",
                      argv[0], argv[1]);
                return(2);
            }
            break;
      case 1:
            fin = stdin;
            break;
      default:
            (void)fprintf(stderr, "Usage: %s [file]\n", argv[0]);
            return(2);
      }
      while ((ch = getc(fin)) != EOF)
            (void)putchar(ch);
      return (0);
}
```

Example 2: Setting internal flags via command arguments

The following example shows how you can set internal flags by means of arguments in a call. The syntax to call the program is:

```
$ prog -opr
```

Depending on which option is specified, `prog` sets the appropriate flags internally. The `getopt()` function in this example (see the “Programmer’s Reference Manual”) is usually used to process arguments.

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

int main(int argc, char *argv[]){
      int oflag = 0;
      int pflag = 0;    /* flags */
      int rflag = 0;
      int ch;
      while ((ch = getopt(argc, argv, "opr")) != -1)
      {
            /* for options present, set flag to 1  */
            /* if unknown options present, print error message  */

            switch (ch){
            case 'o':
                  oflag = 1;
                  break;
            case 'p':
                  pflag = 1;
                  break;
            case 'r':
                  rflag = 1;
                  break;
            default:
                  (void)fprintf(stderr, "Usage: %s [-opr]\n", argv[0]);
                  return(2);
            }
      }
       /* Do other processing controlled by oflag, pflag, rflag. */
      return(0);
}
```

Characters

By default, the data type `char` is treated as `unsigned` by the compiler (see also the `-K uchar` and `-K schar` options on page 41).

The value of an ASCII character is always positive.
The value of '\377' (octal) or '\xFF' (hexadecimal) is thus 255.
If a character constant contains a numeric value that is not included in the ASCII character set, behavior is undefined.

The value of a character constant that contains more than one character (e.g. 'ab') is computed from the ASCII value of the character as a number to the base 256. The first (right) character is multiplied by 1, the second character by 256, the third character by 256 * 256, the fourth character by 256 * 256 * 256.
For example, 'abcd' produces the value 'a' * 256^3 + 'b' * 256^2 + 'c' * 256 + 'd'.

The value of a multibyte character constant in the form L'ab' is identical to the value of a character constant in the form 'ab' in this implementation.

If a character constant contains five or more characters, an error occurs, and no code is generated.
The assignment of `int` to `char` occurs modulo 256.

Multibyte characters

In this implementation, multibyte characters always have a length of 1 byte, and `wchar_t` values are always 32-bit integer values.

Pointers

In the default 32-bit data model (option `-K bit32`), a pointer is represented in 4 bytes and is aligned on a word boundary. The difference between two pointers is of type `int`.

In the 64-bit data model (option `-K lp64`), a pointer is represented in 8 bytes and is aligned on a double-word boundary. In this case, the difference between two pointers is of type `long`.

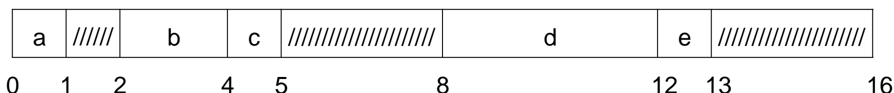
Structures

In structures, components occupy space in the order of their declaration. Each component is aligned in accordance with its type. The structure itself is aligned on the maximum alignment size required for a component. The size of the structure is a multiple of this alignment so that arrays can be constructed from these structures. See also “Alignment of data types” on page 144 and the preprocessor directive `#pragma aligned` on page 150.

Example

```
                          Size:       Alignment:           Offset:

struct {  char  a;        1 byte      byte boundary        0 (word boundary)
          short b;        2 bytes     half-word boundary   2
          char  c;        1 byte      byte boundary        4
          long  d;        4 bytes     word boundary        8
          char  e;        1 byte      byte boundary        12
       };                                                  16 (structure end)
```



Bitfields

Bitfields are stored from left to right in a maximum of 32 bits (one word) when using the 32-bit data model and in a maximum of 64 bits (one double-word) for the 64-bit data model.

Bitfields can be defined as follows:

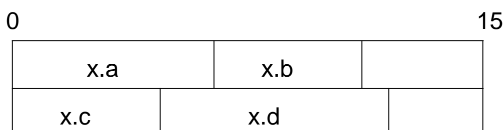
```
int         unsigned int        signed int
long        unsigned long       signed long
short       unsigned short      signed short
char        unsigned char       signed char
```

Bitfields without the `unsigned` or `signed` keyword are represented in accordance with the base type, i.e. `char` as `unsigned` and `int`, `long` and `short` as `signed`. If `signed` or `unsigned` is specified explicitly, the bitfields are represented accordingly. This default behavior can be modified by means of the following options:
`-K schar`, `-K signed_fields_unsigned` and `-K plain_fields_unsigned`
(see page 41ff).

If the bitfield fits in the current byte, half-word, word or double-word, the specified number of bits are placed in it without being aligned; otherwise, the bitfield is aligned on a byte, half-word, word or double-word boundary in accordance with its base type (see example below).

Example

```
struct
{
    unsigned short  a : 7;
    unsigned short  b : 5;
    unsigned short  c : 5;
    unsigned short  d : 8;
} x;
```



Enumerations (enum)

The underlying type for an enumeration type is `int`.

Type qualifier volatile

`volatile` prevents optimization on accessing a variable. This means that instead of using the old contents, new values are always read from storage. For all assignments, including redundant ones, the appropriate value is directly written to storage. In contrast to non- `volatile` objects, which are subject to extensive optimization and are typically held in registers, the implementation guarantees that all references to `volatile` objects will always point to values in storage.
`volatile` is only accepted syntactically in K&R mode.

Storage class register

Variables can be declared as register variables with `register`. This is a hint to the compiler that the variables are used relatively often and should therefore be held in registers. This saves the high overhead of accessing storage when reading and writing such variables. Note, however, that the optimization mechanism of the compiler may ignore such hints and implement variables as register variables in accordance with its own algorithm.

size_t

In this implementation, size_t corresponds to `unsigned int` in the 32-bit data model and `unsigned long` in the 64-bit data model.

ptrdiff_t

In this implementation, ptrdiff_t corresponds to `int` in the 32-bit data model and `long` in the 64-bit data model.

Conversion of data types

- integer --> integer

  When an unsigned integer value is converted to a signed integer type of the same size, the bit pattern is retained. If the value cannot be accommodated, the result corresponds to the subtraction of the largest possible number + 1 from the given size.

  If a conversion of an integer value to a smaller integer type is involved, and the value cannot be accommodated, the bit pattern is retained and the higher-valued bits are truncated.

- floating-point number --> integer

  When a floating-point number is converted to an integer, the number is truncated toward zero.

  Example

  (int)(-1.5) is -1

  (int)(1.5) is 1

  The result is undefined if the floating-point number to be converted is too large to be represented as an integer value.

- integer --> floating-point number

  The conversion of an integer to a floating-point type that cannot accept the correct value is accomplished by rounding.

- floating-point number --> floating-point number

  The conversion of a floating-point number to a smaller floating-point number (e.g. `double` to `float`) is accomplished by rounding.

- integer <--> pointer

  When an integer is converted to a pointer, and vice versa, the bit pattern is not changed (simple reinterpretation).

Sign of division remainder

The remainder of an integral division always has the same sign as the dividend.

Example

(-5) / 2 is -2, (-5) % 2 is -1
5 / (-2) is -2, 5 % (-2) is 1

Logical and arithmetic right shift

If the left operand is unsigned, the right shift is logical (padding of 0 bits); otherwise, arithmetic (padding of signed bits).

Example

(-8) >> 1 is -4

Bitwise operations on signed integer values

Bitwise operations (operators ~, <<, &, ^, and |) are executed as unsigned integers on interpretation; however, the result is signed.

Declarators

Any number of declarators may be used to declare a type.

switch statement

Any number of `case` branches may be used per `switch` statement.

Preprocessor directives

– #include

  The algorithm by which the preprocessor searches specific directories for headers (<name> or “name”) is described under the option `-I` *dir* (see page 37).

  There are no restrictions with respect to the nesting of header files.

– #pragma

  See “#pragma directive” on page 149.

– _ _DATE, _ _TIME_ _

  If the date and time of compilation are not available, these macros are defined as follows:

_ _DATE_ _	"Jan 1 1970"
_ _TIME_ _	"01:00:00"

Size and value ranges for elementary data types

Type	Bit	Value ranges
char	8	0 .. 255
signed char	8	-128 .. 127
short	16	-32768 .. 32767
unsigned short	16	0 .. 65535
int	32	-2147483648 .. 2147483647 (-2^{31} .. $2^{31}-1$)
unsigned int	32	0 .. 4294967295 (0 .. $2^{32}-1$)
long	32 (32-bit model)	same as int
	64 (64-bit model)	-2^{63} .. $2^{63}-1$
unsigned long	32 (32-bit model)	same as unsigned int
	64 (64-bit model)	0 .. $2^{64}-1$
float	32	10^{-75} .. $0.79*10^{76}$
double	64	same as float
long double	124	same as float

Alignment of data types

Data type	Size	Alignment
char, unsigned char, signed char	1 byte	byte boundary
short, unsigned short	2 bytes	half-word boundary
int, unsigned int	4 bytes	word boundary
long, unsigned long	4 bytes (32-bit model) 8 bytes (64-bit model)	word boundary double-word boundary
pointer	4 bytes (32-bit model) 8 bytes (64-bit model)	word boundary double-word boundary
float	4 bytes	word boundary
double	8 bytes	double-word boundary
long double	16 bytes	double-word boundary
Enumerations	Represented as char, short or long with corresponding alignment, depending on limits.	
Arrays	Size and alignment correspond to element type.	
Structures	Size and alignment for individual components based on above rules; overall alignment based on maximum alignment for components.	
Bitfields	If the alignment boundary for the base type is not exceeded, the specified number of bits is created without alignment; otherwise, the bits are aligned in accordance with the base type.	

Implementation-defined limits

Most limits depend on the available system resources (e.g. on virtual memory). Only the following limits are implementation-defined:

Characteristic	**Maximum value**
Number of parameters in a macro definition	2^{24}-1
Number of arguments in a macro call	2^{24}-1
sizeof limit (This limit also applies to static, global and auto objects in the lp64 data model.)	2^{31}

7.3 Extensions to ANSI/ISO C

The extensions described below do not conform to the language features described in the ANSI/ISO standard and could thus result in potentially non-portable source programs if used.

Special character $ and @ in identifiers

If the `-K dollar` and `-K at` options are specified, the “dollar” character $ and the “at” character @ are permitted in internal and external names (see also “Identifiers” on page 136).

main function with three parameters

In addition to the two parameters *argc* and *argv* (see page 136), a third parameter `char *`*envp*`[]` may be declared, where *envp* is a pointer to an array of strings that is supplied with information on the system environment of Reliant UNIX (see also the `exec` functions in the “Programmer´s Reference Manual”).

Scope of functions

`extern` declarations of functions within blocks apply to the entire compilation unit. If multiple `extern` declarations are available for the same function, they are tested for a match.

Write access to string literals

In this implementation, string literals can be overwritten by default. This ensures that the literals do not overlap. Identical literals are stored in separate areas.
The option `-K rostr` can be used to specify read-only access for string literals (see page 61).

Data type long long

The data type `long long` (together with `unsigned long long`) is represented in 8 bytes, with alignment on a double-word boundary. Constants of type `long long` are identified by the suffix `LL` or `ll` following the number. If a constant of type `unsigned long` is too large, it is treated as a constant of type `long long`. A `long long` constant is `unsigned` if it contains the additional suffix `U` or `u` or if it is too large to be represented as `signed long long`.

Conversion of function pointers

The cast operator can be used to convert pointers to objects to pointers to functions, and vice versa. In the case of implicit conversions, warnings are issued by the compiler.

Non-integer bitfields

All integral types can be used as bitfields (see also the section “Bitfields” on page 140). The standard only defines the types `int`, `usigned int` and `signed int`.

The asm keyword

The compiler supports the `asm` statement, which enables the inline substitution of Assembler code in a C program. The general format of an `asm` statement is as follows:

`asm("`*string*`"`[, *expression1*]...[,*expression4*]`);`

Instead of the keyword `asm` , you may also enter `__asm` , since both keywords are semantically identical. The keyword `__asm` must be entered in strict ANSI C mode, since `asm` is not a reserved keyword in this mode.

The string literal *string* can be used to specify any Assembler statement (e.g. an Assembler instruction).

expression1 to *expression4* can be used to pass up to four arguments to the Assembler statement, where each argument must be delimited by a comma. Such arguments may be constant C expressions or C variables, but not another `asm` statement. The arguments are internally held in registers. For each argument, a place-holder in the form `%0` (for the first argument) to `%3` (for the fourth argument) must be provided in the Assembler statement *string*:

Examples of argument passing

```
asm("add %0, %1", sum, val*base)
```

`%0` is replaced by the result of the computed `sum` , and `%1` by the expression `val*base`.

```
asm("mctl %1, %0", f, i)
```

`f` is calculated first; `i`. `%1` is then replaced by `i`, and `%0` by `f`.

The `asm` statement is not always handled like a C statement in a strict sense: if it appears outside functions, it is treated as a declaration, i.e. further declarations may follow.
It is only when the `asm` statement appears within a function or a block that it is treated as a statement, i.e. cannot be followed by further declarations.

The `asm` statement can also be used as an expression, in which case the return value is of type `int`.

Multiple definitions of external variables

If there are so-called “tentative” definitions for the same object (i.e. external declarations of variables without the attribute `extern` or `static`) in several compilation units, these must always be of the same type. Different type declarations for the same external object are not recognized by the compiler. Multiple initializations of external variables will result in errors on linkage.

Empty macro arguments

When calling macros, it is also possible to pass empty arguments.

Example

```
#define F(a,b)   f(a)+f(b);

F(1)    /* produces f(1)+f(); */
F(,1)   /* produces f()+f(1); */
```

Predefined macros

For compatibility reasons, some predefined macros do not begin with the underscore (_) character (see the section “Predefined preprocessor names” on page 78).

Additional preprocessor directives

The following additional preprocessor directives are available: `#line` (old style), `#ident`, `#assert` and `#unassert` (see the section “Preprocessor directives” on page 149).

7.4 Preprocessor directives

This section describes all preprocessor directives available in addition to those in the ANSI/ISO C standard. It also describes the `#pragma` directives which are defined as implementation-dependent in the ANSI/ISO standard and which are accepted by the compiler.

7.4.1 Extensions

#assert directive

`#assert␣`*name*[(*token-sequence*)]

The `#assert` directive can be used to define an assertion. Assertions are independent of macro definitions.

name is the name of the assertion, and *token-sequence* is the value to which the assertion applies.
A single *token* may be one of the following lexical units: name, keyword, constant, string, operator, separator/punctuation character. If no *token-sequence* is specified, the assertion is considered defined as in the case of a symbolic constant, but no value is assigned.

The `#if` statement can be used to test whether an assertion applies to a value:

`#if #`*name*(*non-empty-token-sequence*)

name is the name of the assertion, and *non-empty-token-sequence* is the value to be tested. For example, the following test for the predefined assertion `compiler` would return the value “true”:

```
#if #compiler(CDS++)
```

The predefined assertions are described on page 79.

#unassert directive

`#unassert␣`*name*[(*token-sequence*)]

The `#unassert` directive has the same syntax as `#assert` and can be used to remove any assertion.
If a *token-sequence* is specified, only the assertion for that value is removed; otherwise, i.e. if no *token-sequence* is specified, the entire assertion is deleted.

#ident directive

`#ident␣"`*string*`"`

The `#ident` directive is used for version control.

string can be any arbitrary string that usually contains version control information. The compiler stores this *string* in the `.comment` section of the object file. This section is not loaded into memory when the program is executed.

#line directive (old format)

#␣*digit-sequence*␣[*header-file*]

This directive is identical to the `#line` directive, except for the fact that the keyword `line` is omitted.

7.4.2 #pragma directive

The `#pragma` directive is implementation dependant. The following section describes the `#pragma` directives that are supported by the compiler in the C and C++ language modes.

A description of the C++-specific `#pragma` directives to control template instantiation (`instantiate`, `do_not_instantiate`, `can_instantiate`) can be found in the section “Template instantiation” on page 171.

aligned

`#pragma␣aligned␣`*n*

The `aligned` pragma can be used to align data elements within classes, structures and unions on a larger number of bytes than the default minimum alignment set for the compiler. This pragma can be used in all language modes of the compiler.

n is a number of bytes, which may be specified in steps to the power of 2, up to a maximum of 1024. The permitted entries are thus 1, 2, 4, 8, 16, 32, 64,128, 256, 512 or 1024 bytes.

Notes

For the sake of simplicity, the following notes only refer to structures, but are also analogously applicable to classes and unions.

– Pragmas that specify fewer bytes than the minimum alignment intended for the corresponding data type (see table below) are not permitted and are ignored.

	Minimum alignment (number of bytes)	
Data type	bit32 data model	lp64 data model
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8
float	4	4
pointer	4	8
double	8	8
long double	8	8

– The data element to be aligned may also be a bitfield, in which case the bitfield is created in a new base field with the required alignment.

– In the case of `static` elements, the pragma is ignored.

– The pragma must be placed in the structure definition immediately before the data element to be aligned. Otherwise, it will be ignored.

– If multiple pragmas precede a data element, the one with the largest alignment specification is considered.

– If a pragma precedes the declaration of several structure elements, it is applied to only the first declared element.

– The alignment of a structure is based on the maximum alignment of its elements.

– If a structure appears as an element in another structure, the pragma preceding it applies to the alignment of the entire structure, and not the alignment of its elements.

– A pragma that precedes a structure element that represents an array applies to the alignment of the entire array (i.e. the first array element), and not the remaining array elements.

Example

```
...
class bsp
  {
    int a;                  // Aligned on 4 bytes.
#pragma aligned 16
    int b,c;                // b is aligned on 16 bytes.
                            // c is aligned on 4 bytes!
                            // The maximum alignment of an element
                            // of class bsp is thus equal to 16 bytes.
                            // Class bsp is therefore aligned on
                            // 16 bytes.
    int d;                  // Aligned on 4 bytes.
public:
    double dens;            // Aligned on 8 bytes.
#pragma aligned 4           // Is ignored. Since the maximum alignment
                            // of an element of the structure stru1 is
                            // 8 bytes, stru1 is also aligned on 8 bytes.
                            // A corresponding warning is issued.
    struct
      {
        int istru11;        // Aligned on 4 bytes.
        double dstru12;     // Aligned on 8 bytes.
      } stru1;              // Aligned on 8 bytes (see above).

    struct
      {
        short s1;           // Aligned on 2 bytes.
        short s2;           // Aligned on 2 bytes.
      } stru2;              // Aligned on 2 bytes, since the maximum
                            // alignment of an element equals 2 bytes.
#pragma aligned 4
    char c;                 // Aligned on 4 bytes.
    char c1;                // Aligned on 1 byte.
#pragma aligned 8
    short ar1[16];          // The array is aligned on 8 bytes,
                            // but not the individual array elements.
    ...
  }
...
```

hdrstop

```
#pragma␣hdrstop
```

This pragma sets a header stop point within a header, i.e. a point as of which no further precompiled headers (PCH files) are to be generated or reused.

Example

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

In automatic or manual PCH mode (see the options `-Z pch`, `-Z create_pch`, `-Z use_pch`), a PCH file will be generated or reused only for the headers `xxx.h` and `yyy.h`, but not for `zzz.h`.

More details can be found in the section “Precompiled headers” on page 83ff.

ident

`#pragma␣ident␣"`*string*`"`

This pragma is identical to the preprocessor directive `#ident "`*string*`"` (see page 150).

inline

`#pragma␣inline␣`*name*

This pragma provides the optimizer with a “recommendation” to generate the code for the *name* function inline. This may, however, not be possible for every function. Recursive functions, for example, cannot be expanded inline.

In the C language modes (`cc`/`c89` command), functions marked with `#pragma inline` will be expanded inline only if one of the optimization levels `-F O`*x* (where *x* = 0, 1, 2, or 3) has been turned on. In the C++ language modes (`CC` command), inline expansion occurs independently of the optimization levels. Note that if the option `-F no_inlining` is set, `#pragma inline` is always ignored.

int_to_unsigned

```
#pragma␣int_to_unsigned␣name
```

This pragma was introduced because the result type of some library functions was changed from `int` to `unsigned int` in the transition from K&R C to ANSI/ISO C. The pragma, which only works in K&R mode, instructs the compiler to treat a function *name* with the result type `unsigned` as if its result type were still `int`. This ensures that the program remains compatible with earlier C versions.

The declaration of the function *name* with the result type `unsigned` must appear before the `#pragma` directive, e.g.:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

no_pch

```
#pragma␣no_pch
```

When a source file containing this pragma is compiled, no predefined headers (PCH files) are generated or reused. See also the options `-Z pch`, `-Z create_pch`, `-Z use_pch` (on page 39) and the section “Precompiled headers” (on page 83ff).

pack

```
#pragma␣pack(n)
```

This pragma determines the physical arrangement of structures. *n* is the number 1, 2 or 4 and specifies the smallest possible alignment in bytes for structure components. The default value is 4.

weak

Format 1

```
#pragma␣weak␣name
```

This pragma specifies *name* as a global symbol with the linkage attribute `weak`. This means that the link editor will not issue an error message if *name* is not defined in the program and if a reference to *name* cannot be resolved as a result.

Format 2

`#pragma␣weak␣`*name* = *name2*

name is specified as a global symbol with the linkage attribute `weak` and with the same value as *name2*. This causes the link editor to use the definition of *name2* when resolving a reference to *name* if *name* is not defined. If *name* is defined, then that definition is used. All references to *name2* are always resolved by the link editor by using the definition of *name2*.
This `#pragma` directive allows the use of a user-defined function (on source code level) instead of a standard library function of the same name.
For example, the directive `#pragma weak read = _read` has the following effect:
if a user-specific function named `read` is defined, then that `read()` function will be called on invoking `read()`; otherwise, the standard library function `_read()` is called.

8 C++ language support of the compiler

When the compiler is called with the `CC` command, it operates as a C++ compiler and optionally supports both the Cfront V3.0.3-compatible C++ language scope as well as the scope defined in the ANSI/ISO draft of 1995. The default setting for the `CC` command is extended ANSI C++ (see also the section “Options to select the language mode” on page 34).

The C++ programming language is described in detail in the “The C++ Programming Language, 2nd Edition”, by Bjarne Stroustrup. This manual does not include some of the new ANSI/ISO C++ language features (see the table on page 158). A description of these new features can be found in the addendum to the C++ Reference Manual supplied with CDS++ (see the Release Notice for details).

The following sections describe the vendor-specific and implementation-defined language features of C++. They are intended as a supplement to the C++ language definitions given in the manual by B. Stroustrup and in the addendum.

8.1 Overview of the C++ language modes

In accordance with the different language definitions for C++, the compiler supports three C++ compilation modes:

Cfront C++ mode (option -X d)
The compiler accepts C++ code compatible with Cfront V3.0.3.

extended ANSI C++ mode (option -X w) or
strict ANSI C++ mode (option -X e)
The compiler accepts C++ code based on the ANSI/ISO definition.

The following table contains an overview of the main differences between the various C++ language modes.

Features / language attributes	Cfront 3.0.3	extended ANSI	strict ANSI
Invocation command	CC		
Language mode option	`-X d`	`-X w` (Default)	`-X e`
Reserved keywords [1]			
Exception handling `catch`, `throw`, `try`	no	yes	yes
New ANSI C++ language features [2]		yes	yes
– Runtime type information (RTTI) `typeid`, `dynamic_cast` – Arrays: new/delete `new []`, `delete[]` – Name space `namespace`, `using` – Template parameter `typename` – Constructor type `explicit` – Data type `wchar_t` – Boolean data type `bool`	no		
– Storage class `mutable` – Casting keywords `const_cast`, `reinterpret_cast`, `static_cast`	yes		
`long long`	yes	yes	yes 3)
Thread support	yes	yes	yes
DLL `dlopen / dlclose` [4]	yes	yes	yes
`__STDC__`	==0	==0	==1
`__cplusplus`	==1	==2	==199504L 5)
Object layout and name mangling	compatible	new [6]	new 6)
Data model	bit32, lp64	bit32, lp64	bit32, lp64

Overview of differences between the various C++ language modes

1) Reserved keywords

All keywords listed in the chapter on “C language support” on page 133 are also reserved in the C++ language modes in addition to the following C++-specific keywords.

asm	**explicit**	new	template	**using**
bool [1]	**export** [3]	operator	this	virtual
catch [2]	**false** [1]	private	throw [2]	**wchar_t**
class	friend	protected	**true** [1]	
const_cast	inline	public	try [2]	
delete	mutable	reinterpret_cast	**typeid**	
dynamic_cast	**namespace**	static_cast	**typename**	

Reserved names in C++

The keywords shown in **boldprint** are not reserved in the Cfront C++ mode.

1 The keywords `bool`, `true` and `false` are reserved in the ANSI C++ modes, depending on the setting of the `-K bool` or `-K no_bool` options. `-K bool` is the default.

2 No exception handling is supported in the Cfront C++ mode. The keywords `catch`, `throw` and `try` are nonetheless not freely available and will result in an error message if used.

3 The keyword `export` could change before the ANSI/ISO C++ standard is released.

The following keywords may be used as alternative representations for C operators. The setting of the `-K alternative_token` or `-K no_alternative_token` option determines whether or not they are reserved.
`-K no_alternative_token` (not reserved) is the default in the Cfront C++ mode;
`-K no_alternative_token` (reserved) is the default in the ANSI C++ modes.

<table>
<tr><td>and</td><td>&&</td><td>not_eq</td><td>!=</td></tr>
<tr><td>and_eq</td><td>&=</td><td>or</td><td>||</td></tr>
<tr><td>bitand</td><td>&</td><td>or_eq</td><td>|=</td></tr>
<tr><td>bitor</td><td>|</td><td>xor</td><td>^</td></tr>
<tr><td>compl</td><td>~</td><td>xor_eq</td><td>^=</td></tr>
<tr><td>not</td><td>!</td><td></td><td></td></tr>
</table>

Keyword operators in C++

2) New ANSI C++ language features

The new C++ language features that have been added to the language definition by Stroustrup in accordance with the future ANSI/ISO C++ standard are documented in a language definition addendum supplied with CDS++ (see the Release Notice for details).

3) long long

The data type `long long` is an extension to ANSI C++ and thus results in a warning in strict ANSI C++ mode (see also page 146).

4) dlopen and dlclose

Like C++ V3.1C, CDS++ supports the dynamic loading of shared objects with `dlopen`/`dlclose` in C++ programs (an option originally restricted to C programs).

The descriptions of these functions presented in section on “Functions from special libraries (3X)” in the “Programmer’s Reference Manual” are also applicable to their use in C++ programs. The problems in connection with finalizing global and local static objects when using `dlopen`/`dlclose` are discussed on page 181.

5) __cplusplus

This value will increase in subsequent versions of the compiler and may also change before the final release of the ANSI/ISO C++ standard.

6) Object layout and name mangling

Due to the support for exception handling, RTTI, new[ ] and delete[ ], the object layout and name mangling strategies differ with respect to Cfront V3.0.3.
Consequently, modules generated in the Cfront language mode cannot be linked with modules generated in one of the ANSI C++ modes!

8.2 Implementation-defined behavior based on the ANSI/ISO C++ standard

All of the implementation-defined features based on the ANSI/ISO C standard that have already been described in the chapter “C language support of the compiler” (see page 136ff) are also applicable in the ANSI C++ language modes and are therefore not listed individually here.

Only the implementation-defined C++ language features that extend beyond the scope of the C language are described below.

Linkage of the main function

The `main` function has external C linkage.

Data type bool

The size of type `bool` is defined as `sizeof(bool) == 1`.

reinterpret_cast

The destination object contains the same bit pattern as the expression evaluated by `reinterpret_cast`, but might not be a valid object of the desired type (e.g. `reinterpret_cast<float>(int)`).

The following conversions are possible:

1. A pointer can be explicitly converted to an integral type that is large enough to hold it. Note that pointers have different sizes in different data models (see page 139). Depending on whether the destination type is `signed` or `unsigned`, the value of the result obtained from the conversion may or may not be signed.
2. The value of an integral type can be explicitly converted to a pointer. Note that pointers have different sizes in different data models (see page 139).

Allocation overhead for new arrays

For each allocated array, a structure is reserved at the start of the memory block allocated for that array. This structure contains two `size_t` members, one for the size of the array (in bytes) and one for the number of elements in the array (this field is encoded in order to detect whether the structure has been overwritten).

The asm keyword

The `asm` keyword represents an extension to the ANSI C standard, but is defined as a normal reserved keyword in the ANSI C++ standard. The `asm` statement for generating inline Assembler code is described in the section ”Extensions to ANSI/ISO C” on page 147.

Linkage specification

The supported linkage specifications are “C++” and “ C”.

The default linkage specification is "C++". Names with external C++ linkage are transformed internally by the compiler to enable processing by the link editor (name mangling). Since this internal name mangling process differs in the Cfront C++ mode and in the ANSI C++ modes, Cfront C++ modules cannot be linked with ANSI C++ modules.

In the case of names with external C linkage, no internal name mangling is performed. C linkage can be used to link and call functions written in C or in a language that behaves like C on its name mangling interface.

A pointer to a C function is compatible with a pointer to a C++ function of the same type if the argument types are C-compatible PODs.

Linkage of templates

Only templates with C++ linkage are supported.

Template instantiation

See page 164ff.

Reference types

The reference for an rvalue is bound to a temporary object created by the compiler.

Allocation of non-static data elements of a class

The allocation of memory for these data elements occurs in the strict order of their declarations, i.e. without taking the access specifiers `public`, `private` or `protected` into account.

Bitfields within classes

Bitfields within classes are handled in the same way as bitfields within structures. This applies to the allocation and alignment as well as the handling of “plain” bitfields without the `signed` or `unsigned` attribute.

Constructors and destructors for global and local static objects

See page 178ff.

Exception handling

Thrown exceptions

The memory for exception objects is taken from a preallocated storage, which can be extended by calls to `malloc`.

Handling of exceptions

The stack is not unwound, i.e. no destructors are called for automatic objects if the program exits with `terminate()` due to a missing exception handler (see also the function `unwind_exit()` on page 183).

The unexpected() function

The thrown exception object which causes the `unexpected()` function to fail is destroyed by calling a destructor and is replaced by a new `bad_exception` object.

More details on exception handling can be found on 183ff.

8.3 Template instantiation

The C++ language includes the concept of templates. A template is a description of a class or function that serves as a model for a family of derived classes or functions. For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, or a stack of any user-defined type. These stacks could then be typically written in the source as `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 created as soon as it is required during compilation. The instantiations of template functions and member functions or static data members of a class template (referred to as **template entities** below), by contrast, need not be created immediately. This is due to the following reasons:

– In the case of template entities with external linkage (functions and static data members), it is important to have only one copy of the instantiated template entity throughout the program.

– The ANSI C++ language allows one to write a specialization for a template entity, which means that the user can supply a specific version to be used instead of the instantiation generated from the template for a specific data type.
Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity is available in another compilation unit, it cannot create the instantiation immediately.

– The ANSI C++ language dictates that template functions which are not referenced should not be compiled and should be checked for errors. Consequently, a reference to a template class should not automatically instantiate all the member functions of that class.

Note that some template entities such as inline functions are always instantiated when used.

From the requirements listed above, it is evident that if the compiler is responsible for the entire instantiation (i.e. if the instantiation is done “automatically”), these instantiations can only be performed meaningfully on a program-wide basis. In other words, the compiler cannot make decisions about the instantiation of template entities until it has seen the source code of all compilation units in the program.

The CDS++ compiler provides an instantiation mechanism by which automatic instantiation is carried out at link time (with the aid of a “prelinker”).

More explicit control over the instantiation process is available to the programmer via different instantiation modes that can be selected using options (see below) and by means of `#pragma` directives (see page 171).

8.3.1 Overview of instantiation modes

The compiler provides a total of four instantiation modes, which can be activated with the following options:

`-T auto` (default)
`-T none`
`-T local`
`-T all`

All instantiations requested explicitly with the instantiation directive `template` *declaration* or with the instantiation pragma `#pragma instantiate` *template-entity* are always created by the compiler for each compilation unit in all instantiation modes.

The remaining template entities are instantiated as follows:

`-T auto`
Instantiation is performed across all compilation units by means of a prelinker. This prelinker is activated when an executable file is created with the `CC` command or if the `-y` option (see page 32) is specified. If a shared object (option `-G`) or a prelinked object file (option `-r`) is being generated, no instantiations are done by the prelinker. The principle of automatic instantiation is discussed in detail in the section on "Automatic instantiation" on page 166.

`-T none`
No instantiations other than those requested explicitly are created.

`-T local`
Instantiations are created per compilation unit, i.e. all template entities that are used in a compilation unit are instantiated. The generated functions are given internal linkage. This mode provides a very simple mechanism for getting started with template programming. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly. This method does, however, result in multiple copies of instantiated functions and is therefore not suitable for production use. Note that there may also be problems due to multiple copies of local static variables.

`-T all`
Instantiations are created per compilation unit, i.e. all template entities that are declared or referenced in a compilation unit are instantiated.
All member functions and static variables of a template class are instantiated, regardless of whether or not they are used. Template functions are instantiated even if they have only been declared.

8.3.2 Automatic instantiation

Automatic instantiation (option `-T auto`) is supported by the compiler by default in all C++ language modes. This allows you to compile your source code and link the generated objects without having to worry about how the necessary instantiations are done.

Note that the discussion which follows refers to the automatic instantiation of template entities for which there is no explicit instantiation request (`template` *declaration*) and no `instantiate` pragma.

For each instantiation, the compiler expects a source file that contains both a reference to the required instantiation and the definition of the template entity as well as all types required for the instantiation of that template entity. The latter two requirements can be satisfied by the following methods:

– 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 looks for a source file with the same base name as the `.h` file and a suffix that satisfies the conventions for C++ source file names (see the rules for input file names on page 23ff). This file is then implicitly included by the compiler on instantiation at the end of each compilation unit without a message being issued. See also the section on “Implicit inclusion” on page 169 for details.

– The programmer makes sure that the files that define template entities also contain the definitions of all required types and adds C++ code or instantiation pragmas in those files to request the instantiation of the template entities therein.

The following steps are performed internally during automatic instantiation:

1. Create instantiation information files
The first time that one or more source files are compiled, no template entities are instantiated. For each source file that makes use of a template, an associated instantiation information file is created is no such file exists. An instantiation information file has the suffix `.o.ii`. For example, the compilation of `abc.C` would result in the creation of the file `abc.o.ii`. The instantiation information file must not be modified by the user.

2. Create object files
The created objects contain information on which instantiations could have been created when compiling a source file.

3. Assign template instantiations
   When the object files are linked, the prelinker is called before the actual linking takes place. The prelinker examines the object files, looking for references and definitions of template entities and for added information about entities that could be instantiated. If the prelinker finds a reference to a template entity for which there is no definition in the object files, it looks for an object file that indicates that it could instantiate that template entity. When is finds such a file, it assigns the instantiation to it.
4. Update the instantiation information file
   All instantiations that were assigned to a given file are recorded by name in the associated instantiation information file.
5. Recompile
   The compiler is internally called again to recompile each file for which the instantiation information file was changed.
6. Create new object file
   When the compiler compiles a file, it reads the instantiation information file for that compilation unit and creates a new object file with the required instantiations.
7. Repetition
   Steps 3 to 6 are repeated until all instantiations that are required and can be generated have been created.
8. Linkage
   The object files are linked together.

Once a program has been linked correctly, the associated instantiation information files contain all the names of the required instantiations. From then on, whenever source files are compiled, the compiler will consult the instantiation information file and do the instantiations therein as in a normal compilation run. In other words, except in cases where the set of required instantiations changes, the prelinker will find all required instantiations stored in the object files, so no further instantiation adjustments are needed. This applies even if the entire program is recompiled.

If a specialization of a template entity has been provided somewhere in the program, the prelinker will treat it as a definition. Since this definition will satisfy any references to the template entity, the prelinker will see no need to request an instantiation for that template entity. If a specialization is added to a program that has already been compiled, the prelinker will remove the assignment of the instantiation from the corresponding instantiation information file.

The instantiation information file must not be modified (e.g. renamed or deleted) by the user, except in the following case: if a source file in which a definition was changed and another source file in which a specialization was added are being compiled in sequence in the same compiler run, and the compilation of the first file (with the changed definition) has aborted with an error, the associated instantiation information file must be deleted manually to allow the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, it will issue a message such as:

```
C++ prelinker: A<int>::f() assigned to file test.o
```

Automatic instantiation and libraries

When an executable file is generated with the `CC` command in automatic instantiation mode, the prelinker will do the automatic instantiation only in individual object files (`.o` files), but not in objects that are part of a (`.a` or `.so`) library.

If the `CC` command is used to generate shared objects and store them in a *file*`.so` library (option `-G`), no automatic instantiation is done by the prelinker.

When generating the executable file, libraries that require instances of template entities must either

– already contain these instances (which may be achieved by explicit instantiation and/or the preinstantiation of objects using the `-y` option; see page 32)
– or provide appropriate headers with `can_instantiate` pragmas.

More details can be found in the section on “Libraries and templates” on page 173.

The options `-T add_prelink_files` and `-T rem_prelink_libs` provide a further method of controlling automatic instantiation in connection with libraries. See the section “Template options” on page 48ff for details.

8.3.3 Implicit inclusion

The implicit inclusion of source files is a method of finding definitions of template entities. This method is enabled for the compiler by default (see also the option `-K implicit_include` on page 50) and can be disabled with `-K no_implicit_include`. Implicit inclusion is a logical extension to automatic instantiation, but both mechanisms operate independently and can be individually enabled or disabled. Implicit inclusion can be very useful even when no automatic instantiation is performed.

When implicit inclusion is enabled, the compiler looks for the definition of a template entity in accordance with the following principle: if a template entity is declared in a header file named *basename*`.h` and no definition for it is available in the compiled source code, the compiler will assume that the definition for that template entity is in a source file with the same base name as the header file and with a suffix that is valid for C++ source files (e.g. *basename*`.C`).
Let us assume, for example, that a template entity `ABC::f` is declared in the header file `xyz.h`. If the instantiation of `ABC::f` is requested on compilation, but no definition of `ABC::f` exists in the compiled source code, the compiler will search the directory containing the header file for a source file with the base name `xyz` and a suffix that applies to C++ source files (e.g. `xyz.C`). If such a file exists, it will be treated as if it were included at the end of the source file containing the `#include` directive for `xyz.h`.

To ensure that the file containing the definition of a particular template entity can be found during instantiation, the complete path name of the file with the declaration of the template must be known. This information is not available in files containing `#line` directives. Consequently, implicit inclusion is not possible in such cases.

Implicit inclusion and the `make` *utility*

When working with the `make` utility, implicit inclusions must be taken into account when generating file dependencies. In other words, the object file depends on explicitly included headers as well as implicitly included files with template definitions.

When using the `-M` option, implicit inclusions will be taken into account in automatic instantiation mode only if the instantiation information files have been correctly built.

The following steps are required for this purpose:

1. Compile all source files.
2. Link the program together so that all instantiations are assigned.
3. Generate file dependency lines with the `make` program using the `-M` option (see also page 31).
4. Repeat steps 2 and 3 if the generated template instances have changed.

Control of instantiation assignments

The assignment of instantiations to local object files can be enabled and disabled with the options `-K assign_local_only` and `-K no_assign_local_only` (see also the section “Template options” on page 49).

8.3.4 #pragma directives to control instantiation

The instantiation of individual templates or even a group of templates can be controlled with the following pragmas:

- The `instantiate` pragma causes the template entity that is specified as an argument to be created. This pragma can be used in all instantiation modes. See also the example on page 173.
- The `do_not_instantiate` pragma suppresses the instantiation of the template entity specified as an argument. The typical candidates for this pragma are template entities for which specific definitions (specializations) have been provided. This pragma can be used in all instantiation modes.
- The `can_instantiate` pragma is a hint to the compiler that the template entity specified as an argument can, but need not, be created in the compilation unit. This pragma is required in connection with libraries and is only evaluated in automatic instantiation mode. See also the examples on page 175 and 176.

The following arguments can be specified with the pragmas:

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

When a template class name is specified as an argument (e.g. `A<int>`), the net effect is the same as if the pragma was specified for each member function and for each static data member of that template class. When instantiating an entire class, individual member functions or static data members can be excluded from the instantiation process by using the `do_not_instantiate` pragma.

Example

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

In order to instantiate templates, the appropriate template definitions must be available in the current compilation unit. If an instantiation is requested explicitly with the `instantiate` pragma, and no template definition or only a specific definition (specialization) is available, an error is issued.

Example

```
template <class T> void f1(T); // no body provided
template <class T> void g1(T); // no body provided
void f1(int) { } // specific definition
void main()
{
   int i;
   double d;
   f1(i);
   f1(d);
   g1(i);
   g1(d);
}
#pragma instantiate void f1(int) // error - specific definition
#pragma instantiate void g1(int) // error - no body provided
```

`f1(double)` and `g1(double)`will not be instantiated (due to the missing template definitions), but no error message will be issued in this case during the compilation. The missing template definitions will, however, result in a linker error at link time.

If a member function name (e.g. `A<int>::f`) is specified as a pragma argument, it must not be an overloaded function. 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.

8.3.5 Libraries and templates

Instantiations for template entities (template functions, member functions and static data members of template classes) can be generated in automatic instantiation mode only if the object meets the following conditions:

– It is not part of a library;
– it contains a reference to the template entity or the `can_instantiate` pragma for that template entity,
– and it contains all definitions needed for the instantiation.

A library that requires instances for its implementation must either contain these instances or provide special headers with `can_instantiate` pragmas. These two options are explained individually below.

1. The library contains all required instances

   The main point to be observed here is to ensure that no duplicates are created when using multiple libraries.

   The instantiation of template entities in libraries can be achieved by the following methods:

   a) by using the `-y` option to enable the automatic instantiation mechanism of the prelinker (see page 32)

      Attention

      If multiple libraries that require the same entity are used, there is a potential risk of duplicates being created, since a separate object is not created per entity. This can be avoided by using the option `-T add_prelink_files` (see page 48).

   b) by explicitly instantiating all template entities with the instantiation statement `template` *declaration* or the `instantiate` pragma (see also page 171)

      The main point to be observed here is to ensure that a separate object is created per entity.

      Example

      Consider the following:

      – a library `l.a` with references to the instances `list(Foo1)` and `list(Foo2)`,
      – a header file `listFoo.h` with the declarations of `list`, `Foo1` and `Foo2`
      – and a source file `listFoo.C` with the definitions of `list`, `Foo1` and `Foo2`.

```
// l.h
#ifndef L_H
#define L_H
#include "listFoo.h"
void g();
#endif

// l.C (l.o is element of l.a)
#include "l.h"
void g()
   {
   list(f1);
   list(f2);
   //...
   }

//listFoo.h
#ifndef LIST_FOO_H
#define LIST_FOO_H
template <class T> void list (T t);
class Foo1;
class Foo2;
#endif

//listFoo.C
template <class T> class list (T t) {...};
class Foo1 {...};
class Foo2 {...};
```

Each of the referenced instances are contained in separate objects in the library `l.a`.

```
// lf1.C (lf1.o is element of l.a)
// lf1.C contains an explicit instantiation for list(Foo1)
#include "listFoo.h"
template void list(Foo1);

// lf2.C (lf2.o is element of l.a)
// lf2.C contains a pragma to instantiate list(Foo2)
#include "listFoo.h"
template void list(Foo1);
#pragma instantiate void list(Foo2)
```

2. The header files contain `can_instantiate` pragmas for all required instances.

   Example

   Consider the following:

   – a library `l.a` with a reference to the instance `list(Foo)`,
   – a header file `listFoo.h` with the declarations of `list` and `Foo`
   – and a source file `listFoo.C` with the definitions of `list` and `Foo`.

```
// l.h
#ifndef L_H
#define L_H
#include "listFoo.h"
void g();
#endif

// l.C (l.o is element of l.a)
#include "l.h"
void g()
   {
   list(f);
   //...
   }

//listFoo.h
#ifndef LIST_FOO_H
#define LIST_FOO_H
template <class T> void list (T t);
class Foo;
#pragma can_instantiate list(Foo)
#endif

//listFoo.C
template <class T> void list (T t) {...};
class Foo {...};
```

   The object `user.o` and the library `l.a` are linked together (`CC user.o l.a`).

```
// user.C
#include "l.h"
int f ()
   {
   g();
   }
```

   `user.C` includes `l.h`, which in turn includes `listFoo.h`. Consequently, `user.C` contains the hint that `list(Foo)` can be instantiated.

   Automatic instantiation by the prelinker produces only one instance `list(Foo)` for the program.

Notes

- In order to generate the needed instances, the `can_instantiate` pragma must be contained in a header of the library that will be included by the user programs.
- Special caution is required when using shared libraries!
  A library may consist of separate parts with separate headers which reference instances that are not used in other parts.

  Example

  Consider a shared library `l.so` consisting of two parts. Part 1 contains the object `l1.o` with a reference to `list1(Foo1)`, and part 2 contains the object `l2.o` with a reference to `list2(Foo2)`.

```
// l1.h
#ifndef L1_H
#define L2_H
#include "listFoo1"
void g1 ();
#endif

// l1.C (l1.o is element of part1 of l.so)
#include "l1.h"
class Foo1 f1;
void g1()
   {
   list1(f1);
   //...
   }

// l2.h
#ifndef L2_H
#define L2_H
#include "listFoo2"
void g2 ();
#endif

// l2.C (l2.o is element of part2 of l.so)
#include "l2.h"
class Foo2 f2;
void g2()
   {
   list2(f2);
   //...
   }

//listFoo1.h
#ifndef LIST_FOO1_H
#define LIST_FOO1_H
```

```
template <class T> void list1 (T t);
class Foo1;
#pragma can_instantiate list1(Foo1)
#endif

//listFoo1.C
template <class T> void list1 (T t) {...};
class Foo1 {...};

//listFoo2.h
#ifndef LIST_FOO2_H
#define LIST_FOO2_H
template <class T> void list2 (T t);
class Foo2;
#pragma can_instantiate list2(Foo2)
#endif

//listFoo2.C
template <class T> void list2 (T t) {...};
class Foo2 {...};
```

The object `user.o` and the library `l.so` are linked together.
`user.C` includes `l1.h` and contains only a reference to `g1`.

```
// user.C
#include "l1.h"
int f ()
   {
   g1();
   }
```

The automatic instantiation mechanism of the prelinker generates one instance `list1(Foo1)`. `list2(Foo2)` leads to an unresolved external reference.

One solution for this example would be:

Instead of providing two separate header files `l1.h` and `l2.h`, a single header called `l.h` could be supplied and included by the user program. For example:

```
// l.h
#include "l1.h"
#include "l2.h"
```

8.4 Constructor and destructor calls for global and local static objects

C++ supports the initialization and finalization of objects with constructors and destructors. Constructor and destructor calls can be applied on both dynamic as well as global and local static objects.

Constructors and destructors for dynamic objects

Constructors for dynamic objects are called when objects are created with the `new` operator, on entering functions or local blocks, on processing current parameters, and when temporary objects are created.

Destructors for dynamic objects are called in the reverse order of their construction, e.g. on exiting a function (`return`, `exit`), block, etc., (see also Stroustrup, R.12.).

Constructors and destructors for global and local static objects

When a program contains global objects that need to be initialized, the compiler generates one initialization routine per compiled source file in order to initialize these objects with constructor calls. These initialization routines are called before starting the program, i.e. before the call to the `main` function. The order in which the initialization routines are called is described in the next section.

To finalize global and local static objects, the compiler creates a destruction list, in which the global and local static objects are registered in the order in which the associated definitions were executed. The destructors for the registered objects are called on exiting the program. Destructor calls for global and local static objects occur in the reverse order of their initialization.

Depending on whether you have set one of the ANSI C++ language modes or the Cfront C++ language mode, the compiler will behave differently, as explained in the sections below.

8.4.1 Implementation in ANSI C++ modes

The calls to the initialization routines are placed in the `.init` sections of the objects. Each object contains one `.init` section.

Each program part (an executable or a shared object) contains a destruction list and a call to a function to destroy the objects registered in the destruction list. This function call is contained in the `.fini` section of the program part.

Order of initialization

The order of initialization for global objects depends on the order in which the `.init` sections of the individual objects are connected in the `.init` section of the program part and the order in which these `.init` sections are executed.

When an executable file or a shared object is created, all `.init` sections are combined into a single `.init` section. The order in which the `.init` sections of the individual objects are connected is determined by the order in which these objects are specified by the user on the command line.

Example

```
CC -G -o libX.so objx1.o objx2.o objx3.o
```

The order of the `.init` sections of the individual objects in the shared object `libX.so` is:

1. `.init` section of `objx1.o`
2. `.init` section of `objx2.o`
3. `.init` section of `objx3.o`

The execution order of the `.init` sections of the shared objects and the executable program is as follows:

1. The initialization of shared objects occurs first. This is done in the reverse order in which the objects were registered by the dynamic linker in the dependency list “DT-NEEDED” (also called a “needed list”).

   The shared objects are registered in the DT-NEEDED list as follows:

   a) All shared objects, in the order in which they are specified on the command line.

   b) All required, but still unregistered shared objects of the first object on the command line.

c) All required, but still unregistered shared objects of the second object on the command line,

etc.

Example

```
CC -G -o libX1.so <objects> -l Y1 -l Y2 -l M
CC -G -o libX2.so <objects>
CC -G -o libY1.so <objects> -l Z1 -l X1 -l M
CC -G -o libY2.so <objects> -l Z2 -l Z3 -l X2
...
CC -o a.out main.o -l X1 -l X2 -l M
```

In this case, the DT-NEEDED list of the dynamic binder would be:

`libX1.so, libX2.so, libM.so, libY1.so, libY2.so, libZ1.so, libZ2.so, libZ3.so`

Initialization occurs in the reverse order, i.e. begins with `libZ3.so` and ends with `libX1.so`.

2. This is followed by the initialization for the executable program.

Note

The order of the given object files or libraries must reflect the order in which the objects are initialized. In the above example, `libM.so` would be initialized after the libraries `libZ3.so` to `libY1.so`, but these may be contingent on the initialization of `libM.so`. Note also that a runtime error is produced if the C++ libraries (`libCstd` and `libCrun`) are specified explicitly, since these libraries are added automatically by the compiler.

When a shared object is explicitly opened by the user with `dlopen`, the DT-NEEDED list is created, and the required objects are initialized.

Order of finalization

The order of executing the `.fini` section of the required shared objects and the executable program is the reverse of the order in which their `.init` sections were executed.

The finalization of program parts thus occurs in the reverse order of the initialization (in accordance with the ANSI/ISO C++ draft).

Objects registered in the destruction list of a program are destroyed in the reverse order of their construction.

Deviations from the ANSI C++ standard

The ANSI C++ standard dictates that the finalization of global and local static objects in a program must be completed in reverse order to their construction. The ANSI C++ standard does not, however, specify how shared objects should be handled.

The above requirement is supported by the compiler for program parts (executable and shared objects), but not for a program that uses shared objects. If shared objects are connected to a program with `dlopen` or disconnected with `dlclose`, the specification requiring the destruction of objects in the program in the reverse order of their initialization cannot always be followed. In the case of `dlclose`, the shared object involved must be finalized immediately.

A description of the `dlopen`/`dlclose` functions can be found under the section “Functions from special libraries (3X)” in the “Programmer’s Reference Manual”.

8.4.2 Implementation in Cfront C++ mode

In the Cfront C++ mode, the CDS++ compiler generates initialization routines with names beginning with `__sti_` and uses a global destruction list to register objects for which a destructor is to be called. The name of the function that calls destructors for objects begins with `__std_`.

A tool called `munch` is used to generate a C file, which contains two lists: a list of the generated initialization routines beginning with `__sti_`, and a list of the finalization routines beginning with `__std_`. This C file is compiled and linked into the program.

The initialization routines of the list are called before starting a program, and the finalization routines of the list are called on exiting the program.

Deviations from Cfront

The destruction order of objects compiled with the CDS++ compiler is not the same as for objects compiled with the Cfront compiler C++ V3.1B, since the CDS++ compiler uses a destruction list.

Furthermore, in contrast to C++ V3.1B, CDS++ (and also C++ V3.1C) has an additional feature that allows shared objects to be loaded dynamically with `dlopen`/`dlclose`. The problem of finalization (i.e. the destruction order) that may arise in connection with `dlopen`/`dlclose` is discussed above in the section on “Deviations from the ANSI C++ standard”.

8.4.3 Initialization and exception handling

Exceptions thrown during the initialization of global objects result in a call to `terminate` and thus cause the program to abort without diagnostics. If desired, you can use `set_terminate` to specify some other exception-handling routine to be used in the event of an unforeseen program abort. This function must, however, be called before initializing the global objects.

The CDS++ compiler offers the following solution to specify functions to be used as the "initial current handler":

– You can link your own `__initial_terminate_handler` function of type `terminate_handler` into your program. This function is declared `weak` in the runtime system of the CDS++ compiler. If `__initial_terminate_handler` is defined, the function will then be called as the "initial handler" to terminate exception processing.

– You can also use the functions `__initial_unexpected_handler` and `__initial_new_handler` with the same mechanism. These routines are of type `unexpected_handler` and `new_handler`, respectively.

8.5 Exception handling

Besides the runtime functions defined by the language, the SNI implementation provides two more functions to enable specific exception handling: `unwind_exit` and `get_caught_object_typeid`. These functions are part of the `__SNI_extensions` name space.

unwind_exit - Unwind stack before exiting program

The `unwind_exit` function, like `exit`, is used to terminate a program. In contrast to `exit`, however, a call to `unwind_exit` causes the following additional actions to be performed before the program is exited:

– All `automatic` objects on the runtime stack that have not yet been deleted are destroyed.
– All exception objects that have not been exited are destroyed.

This is followed by the destruction of global objects as in the case of `exit`.

Note that neither `exit` nor `unwind_exit` will destroy objects on the heap that have not been released.

If a destructor called by `unwind_exit` ends with an exception, `terminate` is called implicitly. It is therefore advisable to call `unwind_exit` in the `terminate_handler` as well. This will guarantee that the destructors for all `automatic` and exception objects are eventually called in any case. This cannot result in an endless loop.

The `unwind_exit` function can be called from any part of the program (like `exit`), especially from a `terminate_handler`. Like `exit`, it is supplied with an exit status as an argument (and with the same effect).

The prototype of the `unwind_exit` function is declared in the `<exception>` header:

```
#include <exception>

namespace __SNI_extensions {
      void unwind_exit(int status);
}
```

get_caught_object_typeid - Determine type of caught exception object

The function `get_caught_object_typeid` can be used to determine the type of a caught exception object. It returns the type of the exception object that was most recently caught and was not finished. If the exception object is a pointer type, the type of the object pointed to is returned. If no caught and unfinished exception object exists, an exception of type `bad_typeid` is thrown.

The type of the caught exception object is returned as a reference to a `type_info` object. The class `type_info` and the prototype of the function `get_caught_object_typeid` are declared in the `<typeinfo>` header:

```
#include <typeinfo>

namespace __SNI_extensions {
      const type_info &get_caught_object_typeid(EO_flag_set *pflags);
}
```

If the argument to `get_caught_object_typeid` is non-zero, it is interpreted as an address containing information on whether the caught object is a pointer, and if it is, also the type qualifiers applicable to the object pointed to.

```
EO_NO_FLAGS               : Not a pointer type
EO_IS_POINTER             : Pointer type
EO_POINTER_TO_CONST       : Pointer to a constant object
EO_POINTER_TO_VOLATILE    : Pointer to a volatile object
```

The function `get_caught_object_typeid` can be called in any handler (`terminate_handler`, `unexpected_handler`, `catch(...) {...}`) to obtain information about the type of the caught exception object. This can be useful in diagnosing program runtime errors.

Example

```
#include <typeinfo>
using namespace __SNI_extensions;

void my_terminate_handler()
{
   EO_flag_set    flags;
   const type_info &caught_type = get_caught_object_typeid(&flags);

   cerr << "Program termination caused by exception of type";
   if (flags & EO_IS_POINTER) {
         cerr << (flags & EO_POINTER_TO_CONST    ? "const "    : "")
              << (flags & EO_POINTER_TO_VOLATILE ? "volatile " : "");
   }
   cerr << caught_type.name() << (flags & EO_IS_POINTER ? " *" : "") <<".\n";

   unwind_exit(1);
}
```

8.6 Thread safety

8.6.1 Thread safety with DCE

If the Thread Package of the product DCE (Reliant UNIX) V2.0 is available, full thread safety is turned on automatically for the C++ runtime system on linking a program with the `-K thread` option.

8.6.2 Thread support in the C++ runtime system

This section deals with the methods by which programs can be adapted to thread packages other than DCE.

The C++ runtime routines can be used in three safety modes:

1. No thread safety
2. Restricted thread safety
3. Full thread safety

These modes are controlled by weak external references. The interface for thread support is specified in the `<CDS++/thread.h>` header.

No thread support (default)

The default mode offers no thread safety. It is used when the weak external reference (described below) is not defined. This mode allows applications without threads to run without performance restrictions by preparing the runtime system accordingly.

In this mode, the runtime system reserves 8 Kbytes of storage for exception objects. This enables an exception that may be thrown due to a lack of memory to be handled cleanly in most cases.

Restricted thread safety

In order to activate this mode, a function

```
extern "C" void __thread_init();
```

must be defined to initialize the thread library. The C++ runtime system ensures that this function is called before executing user code (`main` and constructors for global objects).

In addition, another function such as

```
extern "C" void __thread_fini();
```

can be provided to finalize the thread library. This function, if available, is called automatically by the C++ runtime system at the end of the final user code (`main` and destructors for global objects).

Furthermore, either the thread library or the thread application must define a function such as

```
extern "C" void *__thread_lock_op(lock_op, void *);
```

to acquire or release a global lock. This function is called by the C++ runtime system whenever global data needs to be locked. The first argument handed to this function is an operation reflecting the value of the enumeration type `lock_op`, which is defined in the `<CDS++/thread.h>` header as follows:

```
enum lock_op {

    LOCK_CREATE,      // Create a global lock object
    LOCK_ACQUIRE,     // Acquire a lock on that object
    LOCK_RELEASE,     // Release a lock on the object
    LOCK_DESTROY      // Destroy the lock object

};
```

In the case of `LOCK_CREATE`, `__thread_lock_op` should ignore the second argument and return a pointer to the created lock object. In the other operations, such a pointer to a lock object is passed to the function as the second argument.
For `LOCK_CREATE`, `LOCK_ACQUIRE` and `LOCK_RELEASE`, the function is expected to return the pointer to the lock object as its result or zero if an error occurs.
For `LOCK_DESTROY`, the pointer to the lock object that could not be destroyed is expected as a return value in the event of an error; otherwise, zero.

If an exception is thrown in a thread, but is not handled by the same thread, the current thread is terminated (by calling `terminate`). No thread catches the exceptions of another thread. A `rethrow` call reactivates the last caught and unfinished exception of the current thread.

Finally, either the thread library or the thread application must define a function such as

```
extern "C" void *__thread_specific_op(specific_op, void *, void *);
```

to enable thread-specific data to be addressed. The first argument handed to the function is an operation reflecting the value of the enumeration type `specific_op`, which is defined in the `<CDS++/thread.h>` header as follows:

```
enum specific_op {
```

```
  SPECIFIC_CREATE,   /* Create a key for thread-specific
                        data. */
  SPECIFIC_SET,      /* Set a pointer to thread-specific data
                        under this key. */
  SPECIFIC_GET,      /* Get a pointer to the data of the current
                        thread for this key. */
};
```

For `SPECIFIC_CREATE`, `__thread_specific_op` should ignore the third argument and return a key for a class of thread-specific data (or the value zero on error). If the second argument is non-zero, it is the address of a function. In this case, `__thread_specific_op` should cause the thread library to call this function at the end of each thread with the specific thread value that was assigned to the key as the function argument.
For `SPECIFIC_SET`, the pointer that is passed as the third argument is expected to be entered under the key that is passed as the second argument.
For `SPECIFIC_GET`, `__thread_specific_op` should ignore the third argument and return the pointer that is entered under the key (which was passed as the second argument) for the current thread.

The following restrictions apply in this mode:

- The functions `set_new_handler`, `set_unexpected` and `set_terminate` affect all threads, i.e. they replace the `new_handler`, `unexpected_handler` and `terminate_handler` of all threads. The function `unwind_exit` causes the entire process to terminate with `exit`, and the default `terminate_handler` ends the entire process with `abort`.
- The runtime function `uncaught_exception` may be slow, and there is no preallocated storage for exception objects. This means that exceptions thrown due to a lack of memory might not be handled cleanly: if there is not enough memory available to copy the exception object, the `terminate` function is called.

This mode is only recommended as the first step to run a thread application if the thread library if not compatible with the interface described in the section below. If you are implementing a thread library, it is better to avoid using this mode.

Full thread safety

Two functions are defined by the runtime system in the `<CDS++/thread.h>` header

```
typedef void *thread_context_handle;
extern "C" thread_context_handle __create_thread_context(size_t);
extern "C" void __destroy_thread_context(thread_context_handle);
```

to create or destroy a thread context. In order to use this mode, the thread library must allocate a thread context for every thread, except for the main thread, and must define the function

```
extern "C" thread_context_handle __get_thread_context();
```

to return a handle to the context created for the current thread or the value zero for the main thread.

In this mode, the functions `set_new_handler`, `set_unexpected` and `set_terminate` affect only the thread from which they were called. The function `uncaught_exception` is fast, and storage is preallocated for exception objects. The amount of space to be allocated can be defined in the call to `__create_thread_context`; the value 0 defaults to 8 Kbytes per thread.

If a request for memory fails when an exception is thrown from a thread or if a thread ends in a call to `unexpected` or `terminate` during exception handling, and no `set_new_handler`, `set_unexpected` or `set_terminate` is called by that thread, the appropriate default handling defined by the C++ standard is executed.

This means, in particular, that `terminate` will call `abort`. Alternatively, the thread application or the thread library can define a function

```
extern "C" void __thread_abort();
```

to abort only the current thread. If a `__thread_abort` function has been defined, the runtime system will use it as the `default_terminate_handler` for every thread except the main thread.

When the function `unwind_exit` is called, it normally calls `exit` after unwinding the stack. This behavior is, however, not suitable if `unwind_exit` is called from the exit routine of the thread. For this reason, the runtime system allows the thread application or the thread library to define a function

```
extern "C" void __thread_exit(int);
```

to exit only the current thread. If a `__thread_exit` function has been defined, `unwind_exit` will call that function instead of `exit` for all threads, except the main thread. This ensures that only the current thread is terminated and that no global data is destroyed. `unwind_exit` passes its argument as an exit status to `__thread_exit`.

8.7 Extensions to ANSI/ISO C++

The extensions described below are accepted in all C++ language modes, except when the option `-R strict_errors` has been set in the strict ANSI C++ mode (see page 72). Note that all extensions to ANSI/ISO C see page 146ff) are also supported in the C++ modes.

Declarations and definitions in classes

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

```
class A {

   friend B; // ANSI requires friend class B;
};
```

- Constants of scalar type may be defined within a class:

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

};
```

- A qualified name may be used in the declaration of a class member:

```
struct A {
   int A::f(); // ANSI requires int f();

};
```

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a default assignment operator, i.e. such a declaration blocks the implicit generation of a copy assignment operator. For example:

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

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

Operator functions

`operator()()` functions can have default argument expressions. A warning is issued in this case.

Preprocessor

The preprocessor macro `c_plusplus` is defined in addition to the standard-compliant macro `__cplusplus`.

8.8 Variations in the Cfront C++ mode

The behavior of the compiler in the Cfront C++ mode is compatible with Cfront C++ V3.0 and later versions, i.e. the compiler supports many of the corresponding function attributes and specific features. The Cfront C++ mode is supported so that existing code containing extensions to Cfront versions as of V3.0 can be compiled without manual intervention. It does not guarantee full compatibility with the C++ compiler V3.1B/C.

Consequently, note that if a program produces an error when compiled with the Cfront C++ V3.1B/C compiler, it is possible that the CDS++ compiler may produce a different error or no error at all in the Cfront C++ mode. Some of the special aspects to be noted for the Cfront C++ mode are described below.

- `const` qualifiers on the `this` parameter may be dropped in some contexts, as in this example:

  ```
  struct A {
     void f() const;
  };

  void (A::*fp)() = &A::f;
  ```

  This is actually a safe operation. A pointer to a function of type `const` may be put into a pointer to a non-`const` type, since a call using the pointer is permitted to modify the object, and the function pointed to will actually not modify the object. An assignment in the reverse direction would not be safe.

- Conversion operators specifying conversion to `void` are allowed.

- A `friend` declaration may introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in the ANSI C++ mode, but is also allowed to introduce a new type name in the Cfront mode.

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

- The third operator of the `?` operator is a conditional expression instead of an assignment expression.

- A reference to a pointer type may be initialized from a pointer value without using a temporary variable even if the reference pointer type has supplementary type qualifiers in addition to those present in the pointer value. For example:

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

- A reference variable may be initialized with 0.

- Since the accessibility of types is not checked in the Cfront mode, access errors for types are issued as warnings instead of errors.

- When calling overloaded functions, a null pointer must be written as a string in the form "0". Other notations such as `const` variables with the value 0 or constants in the form '\0' are not interpreted as a null pointer by the compiler in the case of overloaded functions.

- No warning is issued when an `operator()()` function has default argument expressions.

- An alternate form of declaring pointer-to-member-function variables is supported. This is illustrated in the example below:

```
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
```

In this example, `T` names 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 non-standard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to a single standard pointer-to-member declaration in the form:

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

A non-standard 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. For declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration.

- `protected` class members are not checked when the address of a `protected` member is specified.

```
class B { protected: inti; };
class D : public B {void mf(); };

void D::mf() {
   int B::* pmi1 = &B::i;      // error in ANSI mode, OK in Cfront mode
   int D::* pmi2 = &D::i;      // OK
}
```

  Note that the checking of `protected` class members for other operations (i.e. everything except the declaration of pointer-to-member addresses) is handled as defined by the standard in Cfront mode.

- Constructor and destructor calls for global objects and local static objects

  See page 181ff.

- The destructor of a derived class may implicitly call the `private` constructor of a base class. This is an error in the ANSI C++ mode, but is reduced to a warning in the Cfront C++ mode. 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);
  ```

  According to the standard, `int(d)` is interpreted as a parameter declaration (with redundant parentheses), which means that `x` is a function. In the Cfront C++ mode, `int(d)` is interpreted as an argument, so `x` is a variable.

  Note that the declaration `A(x2);` is also interpreted differently in the Cfront C++ mode as compared to the standard. The standard dictates that it should be interpreted as the declaration of an object named `x2`, but in the Cfront C++ mode it is interpreted as a `cast` of `x2` to type `A`.

  A similar deviation from the standard can be seen in the interpretation of the following declaration:

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

  According to the standard, this declares the function `xyz`, which takes a parameter that is a function without arguments and returns an `int`. In the Cfront mode, this is interpreted as the declaration of an object that is initialized with the value 0.

- Bitfields

  A named bitfield may have a size of 0. The declaration is treated as if no name were declared.

  Plain bitfields, i.e. bitfields declared with the type `int`, are always `unsigned`.

- The name for a type specifier may be a `typedef` name that is a synonym for a class name:

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

- No warning is issued on duplicate size and sign (`signed` or `unsigned`) type specifiers:

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

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

```
struct A {
   virtual void f() {}
   A() {}
   ~A() {}
};

struct B : public A {
   B() {}
   ~B() {f();}             // Should call A::f according  to ARM  12.7
};

struct C : public B {
   void f() {}
} c;
```

  In the Cfront mode, `B::~B` calls the function `C::f`.

- An extra comma is allowed after the last argument in an argument list:

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

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

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

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (like C structures), and the destructor is not called on the “copy”. In ANSI mode, the class object is copied to a temporary object; the address of the temporary object is passed as the argument, and the destructor is called on the temporary object after the call returns. 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 constructors or destructors). However, a warning is issued.

- If an unnamed class appears in a `typedef` declaration, the `typedef` name may be used as the class name.

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

- A `typedef` name may be used in an explicit destructor call:

```
struct A { ~A(); };
typedef A B;
int main() {
   A *a;
   a->~B();             // Permitted in Cfront mode

}
```

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

```
class A {
   void f(int) const;
   static void f(int);    // no error in Cfront mode
}
```

9 Appendix: Overview of options (in alphabetic order)

Option	Category	Page
−−	General options	29
−A	Preprocessor	37
−B do_jmp	Link editor	66
−B dynamic	Link editor	65
−B static	Link editor	65
−B symbolic	Link editor	65
−C	Preprocessor	37
−c	Compilation phases (object code)	30
−D *name*[=*value*]	Preprocessor	37
−d n	Link editor	66
−d y	Link editor	66
−E *name*	Compilation phases (preprocessor)	31
−e	Link editor	66
−F afep	Optimization	55
−F big_got	Object generation	55
−F Blimit	Optimization	55
−F default_inlining	Optimization	53
−F default_xbb	Optimization	53
−F feedback_summary	Optimization	54
−F G*n*	Optimization	55
−F I	Optimization	56
−F hw_br_predict	Optimization	56
−F i	Optimization	57
−F inline_limit	Optimization	57
−F loopunroll	Optimization	57
−F no_br_elimination	Optimization	53

Option	**Category**	**Page**
−F no_feedback_uopt	Optimization	53
−F no_inlining	Optimization	53, 57
−F no_positioning	Optimization	53
−F no_splitting	Optimization	53
−F noxbb	Optimization	57
−F O2	Optimization	51
−F O3	Optimization	52
−F O4	Optimization	52
−F O5	Optimization	52
−F Olimit	Optimization	57
−F profdata	Optimization	54
−F profdir	Optimization	54
−F select_ucode	Optimization	53
−F sortedges	Optimization	58
−F space_time	Optimization	58
−F U	Optimization	58
−F ucode	Optimization	58
−F unrolllimit	Optimization	58
−F X, −F X4	Optimization	54
−G	Link editor	66
−g	Object generation	59
−H	Preprocessor	37
−h name	Link editor	66
−I *dir*	Preprocessor	37
−K [no_]alternative_tokens	C and C++ frontend	43
−K ansi_cpp	Preprocessor	38
−K [no_]assign_local_only	C++ frontend (templates)	49
−K [no_]at	C and C++ frontend	41
−K bit32	C and C++ frontend	42
−K [no_]bool	C++ frontend (general)	45
−K [no_]dollar	C and C++ frontend	41
−K [no_]dual	Object generation	59
−K force_vtbl	C++ frontend (general)	44

Option	Category	Page
−K [no_]implicit_include	C++ frontend (templates)	50
−K [no_]instantiation_flags	C++ frontend (templates)	50
−K kr_cpp	Preprocessor	38
−K [no_]link_startups	Link editor	67
−K [no_]link_stdlibs	Link editor	67
−K [no_]longlong	C and C++ frontend	43
−K long_preserving	C and C++ frontend	42
−K lp64	C and C++ frontend	42
−K [no_]mb	C and C++ frontend	41
−K mips1	Object generation	59
−K mips2	Object generation	59
−K mips3	Object generation	59
−K mips4	Object generation	59
−K new_for_init	C++ frontend (general)	46
−K no_pic	Object generation	60
−K normal_vtbl	C++ frontend (general)	44
−K old	Object generation	60
−K old_for_init	C++ frontend (general)	46
−K [no_]old_spezialization	C++ frontend (general)	46
−K pic	Object generation	60
−K plain_fields_signed	C and C++ frontend	41
−K plain_fields_unsigned	C and C++ frontend	41
−K [no_]roconst	Object generation	61
−K [no_]rostr	Object generation	61
−K schar	C and C++ frontend	41
−K selpic	Object generation	61
−K signed_fields_signed	C and C++ frontend	41
−K signed_fields_unsigned	C and C++ frontend	41
−K suppress_vtbl	C++ frontend (general)	44
−K [no_]thread	Object generation	62
−K uchar	C and C++ frontend	41
−K [no_]ucode_libraries	Optimization	58
−K unsigned_preserving	C and C++ frontend	42

Option	**Category**	**Page**
−K [no_]using_std	C++ frontend (general)	45
−K [no_]verbose	General options	27
−K [no_]wchar_t_keyword	C++ frontend (general)	45
−l *archive*	Link editor	24, 68
−L *dir*	Link editor	24, 68
−M	Compilation phases (preprocessor)	31
−N cif	CIF	75
−N output	Listings	73
−N shortsource	Listings	73
−N shortxref	Listings	74
−O	Optimization	51
−o *output_destination*	General options	27
−P	Compilation phases (preprocessor)	31
−p	Object generation	62
−q f	Object generation	62
−q l	Object generation	62
−Q n	Object generation	64
−q p	Object generation	62
−Q y	Object generation	64
−r	Link editor	69
−R error	Compiler messages	71
−R limit	Compiler messages	71
−R min_weight	Compiler messages	71
−R [no_]msg_id	Compiler messages	71
−R [no_]msg_wrap	Compiler messages	71
−R note	Compiler messages	71
−R [no_]show_column	Compiler messages	72
−R strict_errors	Compiler messages	72
−R strict_warnings	Compiler messages	72
−R suppress	Compiler messages	72
−R use_before_set	Compiler messages	72
−R warning	Compiler messages	71
−S	Compilation phases (Assembler code)	31

Option	**Category**	**Page**
−s	Link editor	69
−T add_prelink_files	C++ frontend (templates)	48
−T all	C++ frontend (templates)	47, 165
−T auto	C++ frontend (templates)	47, 165
−T local	C++ frontend (templates)	47, 165
−T max_iterations	C++ frontend (templates)	47
−T none	C++ frontend (templates)	47, 165
−T rem_prelink_libs	C++ frontend (templates)	49
−U *name*	Preprocessor	39
−u *name*	Link editor	69
−V	General options	28
−v	Compiler messages	72
−w	Compiler messages	72
−W l,...	Link editor	69
−X a	Language mode (C)	34
−X c	Language mode (C)	35
−X d	Language mode (C++)	35
−X e	Language mode (C++)	36
−X t	Language mode (C)	34
−X w	Language mode (C++)	36
−y	Compilation phases (prelinker)	32
−Y F,...	General options	28
−Y I,...	Preprocessor	39
−Y P,...	Link editor	70
−Y S,*dir*	Link editor	70
−Z create_pch	Preprocessor	39, 89
−Z pch	Preprocessor	39, 87
−Z pch_dir	Preprocessor	40
−Z [no]pch_messages	Preprocessor	39
−Z use_pch	Preprocessor	39, 89

Abbreviations

ABI	System Application Binary Interface, MIPS Processor Supplement
ANSI	American National Standards Institute
API	Application Program Interface
DBX	Debugger for UNIX
CIF	Compiler Information File
GOT	Global Offset Table
IEEE	Standard for binary floating-point arithmetic
ISO	International Standardization Organisation
K&R	Kernighan & Ritchie
OO API	Object-Oriented API
PCC	Portable C Compiler
PCH	Precompiled Header
PIC	Position Independent Code
POD	Plain Old Data type

Related publications

Manuals from Siemens Nixdorf Informationssysteme AG

[1] Reliant UNIX 5.43
Programmer´s Reference Manual

Contents
Description of the commands for program development, C library functions and system calls, and a description of a number of header files and C-specific file formats.

[2] SINIX V5.41
Programmer´s Guide: Internationalization - Localization

Contents
Guidelines describing the process of developing internationalized software and generating localized databases. This manual also deals with the use of internationalized software on the user level.

[3] **DBX V2.3**
Symbolic Debugger for C, C++ and FORTRAN
User Guide

Contents
Description of the DBX interactive symbolic debugger with its character-based and graphical user interface and special characteristics when working under SoftBench and debugging DCE threads.

[4] Reliant UNIX 5.43
(RM200, RM300, RM400, RM600)
Commands. User´s Reference Manual
Volume1: A-K
Volume2: L-Z

Contents
These manuals are reference works describing the user commands A-K (Volume1) and L-Z (Volume2) of the Reliant UNIX V5.43 operating system.

[5] C-DS
C V1.0 (SINIX)
Guide to Tools for Programming in C
User Guide

Contents
Chapters 3 to 8 of this User Guide should also be useful to programmers working with the CDS++ compiler. It contains over 200 pages of detailed descriptions and numerous illustrative examples for the following utility routines:
– cscope - interactive program examination
– lex - generating a scanner
– lint - generating C-Programs
– m4 - macro processor
– make - managing group files
– SCCS - Source Code Control System
– yacc - generating a parser
Approximately 50 pages in chapter 2 of this manual (“The C compilation system”) are no longer valid when programming with the CDS++ compiler.

[6] **Standard C++ Library V1.2**
User´s Guide and Reference

Contents
Problem-oriented description and reference manual for ANSI-C++ libraries Strings, Containers, Iterators, Algorithms and Numerics.

[7] **Tools.h++ V7.0**
User´s Guide

Contents
Problem-oriented description of the C++ Tools.h class libraries.

[8] **Tools.h++ V7.0**
Class Reference

Contents
Reference manual for the C++ Tools.h class libraries.

[9] **Cfront C++ Library**
Reference Manual

Contents
Classes, functions and operations for complex math and stream-oriented I/O (compatible with Cfront V3.0.3).

Ordering Manuals

Please apply to your local office for ordering the manuals.

Other References

See also the section “Documentation for CDS++ and additional references” on page 2ff.

[10] **The C Programming Language**
2nd Edition - ANSI-C
by Brian W. Kernighan and Dennis M. Ritchie

Contents
Introduction to C, problem-oriented description of the C language elements with a large number of examples, reference section (C language description), C solution

[11] **The C++ Programming Language**
(2nd Edition)
by Bjarne Stroustrup

Contents
This standard work by C++ originator Bjarne Stroustrup includes an introduction to C and C++ with a large number of examples, three chapters on software development using C++ and a complete reference manual.

Index

#assert directive 149
#ident directive 150
#include directive 143
#line directive 150
#pragma directive 143, 150
    for template instantiation 171
#unassert directive 149
--, end input of options 29
.a file 24
.C file 24
.c file 24
.cif file 75
.I file 24, 31
.i file 24, 31
.lst file 73
.mk file 27
.o file 24, 30
.pch file 39, 87
.s file 24, 31
.so file 24
__cplusplus 158
__DATE__ 143
__STDC__ 79
__STDC_VERSION__ 79
__TIME__ 143

32-bit data model
    -bit32 option 42
64-bit data model
    -lp64 option 42

A
-A 37
-A - 37

a.out 22
ABI MIPS 60
address space
  virtual 111
aligned, #pragma directive 150
alignment of data types 144
  aligned pragma 150
ANSI C mode
  extended 34, 128
  strict 35, 128
ANSI C++ mode
  extended 36, 157
  strict 36, 157
argc, parameter for the main function 136
argv, parameter for the main function 136
arithmetic conversions 134
asm, generate inline Assembler code 147, 162
Assembler 11
Assembler inline code, asm statement 147
Assembler source code 31
assertions (see preprocessor assertions) 79

B
-B do_jmpopt 66
-B dynamic 24, 65
-B static 24, 65
-B symbolic 65
backend, compiler component 11
basic block 91
bitfields 140, 147, 163
bitwise operations 143
bool, C++ data type 161
branch elimination 104

C
-C 37
-c 30
C language modes 128
  cc/c89 command 34
C language support of the compiler 127
  extensions to ANSI/ISO C 146
  implementation-defined behavior 136
  overview of the C language modes 128
  preprocessor directives 149

C++ language modes 157
C++ language support of the compiler 157
    constructor and destructor calls 178
    exception handling 183
    extensions to ANSI/ISO C++ 190
    implementation-defined behavior 161
    overview of the C++ language modes 157
    template instantiation 164
    thread safety 186
    variations in the Cfront C++ mode 192
C++ libraries 14
C++ mode, CC command 35
C/C++ compiler
    general characteristics 8
    structure 10
c89 command 21
cache 104
can_instantiate, #pragma directive 171
CC command 21
cc command 21
CDS++, C/C++ development system 7
Cfront C++ language mode 157, 192
Cfront C++ library 17
Cfront C++ mode 35
char, C data type 139
character constant 139
CIF information 75
classes 162
code generator 11
command
    ldd 125
    lprof 62
    nm 120
    prof 62
    size 112
compilation 10
compiler (see C/C++ compiler) 10
compiler messages 80
compiler options (see options) 26
constructors 178
conversions 142
    arithmetic 134
cross-reference listing 74

D
-D 37
-d n 66
-d y 66
data segment 11, 112
    minimizing 118
data types in C
    alignment 144
    size and value ranges 144
DBX 19, 59
debugger DBX 19
debugger option 59
declarators, number 143
destructors 178
diagnostic messages of the compiler 71, 80
digraph sequences 133
directive
    #assert 149
    #ident 150
    #line 150
    #pragma 150
    #unassert 149
    preprocessor 149
directory
    dynamic linker 123
    -I option 37
    -L option 69
    link editor 121
    searched for header files 38
    standard (for libraries) 68
division remainder, sign 142
dlclose 158, 160, 181
dlopen 158, 160, 181
do_not_instantiate, #pragma directive 171
dual object file 30, 59, 97
dynamic library 65, 116
dynamic linking 12, 66, 111, 112
    directories 123

E
-E 31
-e 66
enum 141
enumerations (enum) 141

environment variable
LANG 81
LD_BIND_NOW 125
LD_LIBRARY_PATH 122
LD_RUN_PATH 123
NLSPATH 81
envp, parameter for the main function 146
exception handling 163, 183
exit status, cc/c89/CC command 25
extern "C" 162
extern "C++" 162
external reference 12

F
-F afep 55
-F big_got 55
-F Blimit 55
-F default_inlining 53
-F default_xbb 53
-F feedback_summary 54, 110
-F Gn 55
-F hw_br_predict 56
-F I 56
-F i 57
-F inline_limit 57
-F loopunroll 57
-F no_br_elimination 53
-F no_feedback_uopt 53
-F no_inlining 53, 57
-F no_positioning 53
-F no_splitting 53
-F noxbb 57
-F O2 51
-F O3 52
-F O4 52, 103
-F O5 52, 103
-F Olimit 57
-F profdata 54, 63, 108, 110
-F profdir 54, 63, 108, 110
-F select_ucode 53
-F sortedges 58
-F space_time 58
-F U 58
-F ucode 58

-F unrolllimit 58
-F X 54
-F X4 54
feedback optimization 52, 62, 103
files, cc/c89/CC command 76
freestanding environment 136
frontend options
  C++-specific 44
  common options in C and C++ 41
frontend, compiler component 10
function
  getopt() 138
function entry 55
function inlining 102
  feedback-directed 104

G
-G 66
-g 59
getopt() 138
global data 55
global optimization measures 51
global optimizations 99
gp register 55

H
-H 37
-h 66
hdrstop, #pragma directive 84, 90, 153
header file
  precompiled (see PCH file) 83
  search for 37
header stop point, PCH file 84
hosted environment 136

I
-I 37
ident, #pragma directive 153
identifier 136, 146
implementation-defined behavior
  C language mode 136
  C++ language mode 161
inline generation of functions 52, 56
inline, #pragma directive 153

input file, cc/c89/CC command 23
instantiate, #pragma directive 171
instantiation of templates 164
int_to_unsigned, #pragma directive 154
interactive device 136
intermediate code 10
intermediate code generator 10
ISO C mode (see ANSI C mode) 34

K
-K alternative_tokens 43
-K ansi_cpp 38
-K assign_local_only 49
-K at 41
-K bit32 42
-K bool 45
-K dollar 41
-K dual 59, 97
-K force_vtbl 44
-K implicit_include 50
-K instantiation_flags 50
-K kr_cpp 38
-K link_startups 67
-K link_stdlibs 67
-K long_preserving 42
-K longlong 43
-K lp64 42
-K mb 41
-K mips1 59
-K mips2 59
-K mips3 59
-K mips4 59
-K new_for_init 46
-K no_alternative_tokens 43
-K no_assign_local_only 49
-K no_at 41
-K no_bool 45
-K no_dollar 41
-K no_dual 59
-K no_implicit_include 50
-K no_instantiation_flags 50
-K no_link_startups 67
-K no_link_stdlibs 67
-K no_longlong 43

-K no_mb 41
-K no_old_specialization 46
-K no_pic 60
-K no_roconst 61
-K no_rostr 61
-K no_thread 62
-K no_ucode_libraries 58
-K no_using_std 45
-K no_wchar_t_keyword 45
-K normal_vtbl 44
-K old 60
-K old_for_init 46
-K old_specialization 46
-K pic 60
-K plain_fields_signed 42
-K plain_fields_unsigned 42
-K roconst 61
-K rostr 61
-K schar 41
-K selpic 60
-K signed_fields_signed 41
-K signed_fields_unsigned 41
-K suppress_vtbl 44
-K thread 62
-K uchar 41
-K ucode_libraries 58
-K unsigned_preserving 42
-K using_std 45
-K verbose 27
-K wchar_t_keyword 45
K&R C mode 34, 128
-K, general input rules 23
keyword operators in C++ 159

L
-l archive_code 24, 68
-L dir 24, 69
LANG 81
language support of the compiler
    C 127
    C++ 157
LD_BIND_NOW 125
LD_LIBRARY_PATH 122
LD_RUN_PATH 123

ldd, command 125
libdl 113
library 12
  Cfront C++ 17
  dynamic 65, 116
  libdl 113
  shareable 111
  shared 12
  standard C++ 14
  static 12, 65, 116
  Tools.h++ 18
line profiling 62
link editor 12, 111
  directories 121
  dynamic 111
  undefined symbol 111
link editor options 65
linkage
  of templates 162
  of the C and C++ standard libraries 114
linkage specification in C++ 162
linking
  dynamic 12, 66, 112
  static 12, 66, 111
listing generators 10
listings
  cross-reference 74
  source program 73
long long, C data type 146
loop unrolling 57, 99
lprof command 62

M
-M 31
machine instruction 11
macro arguments, empty 148
main function 136, 146
  linkage in C++ 161
make command 31
make utility 169
message catalogs 81
message output
  options 71
  structure of compiler messages 80

mon.out 62
multibyte character constant 139
multibyte characters 139
multiple definitions of external variables 148

N
-N cif 75
-N output 73
-N shortsource 73
-N shortxref 74
name mangling in C++ 160
new array 161
NLSPATH 81
nm, command 120
no_pch, #pragma directive 154

O
-O 51
-o output_destination 27
object
    shareable 111
object code 30
object file 30
object generation
    options 59
object layout in C++ 160
operands, cc/c89/CC commands 23
optimization 91
optimization options 51
optimizer 11
options 22, 26
    C++-specific 44
    compilation phases 30
    frontend in C and C++ 41
    general 26
    input rules 22
    language modes 34
    link editor 65
    listings and CIF information 73
    message output 71
    object generation 59
    optimization 51
    overview in alphabetic order 199
    preprocessor 37

templates 47

P
-P 31
-p 62
pack, #pragma directive 154
paging 120
parser 10
PCH file 83
options 39
PIC 60
pointer 139
position independent code (PIC) 60
pragmas 150
for template instantiation 171
precompiled header (see PCH file) 39
precompiled header file (see PCH file) 83
predefined preprocessor assertions, cc/c89/CC command 79
predefined preprocessor macros, cc/c89/CC command 78
prelinker, automatic template instantiation 165
preprocessor 10
preprocessor assertions, predefined 79
preprocessor directives 143, 149
preprocessor macros, predefined 78
preprocessor options 37
preprocessor phase 31
procedure positioning 104
procedure splitting 104
prof command 62
profiling directory 63
profiling executable 106
profiling raw data file 62
prototyping 134
ptrdiff_t 142

Q
-q f 62, 106
-q l 62
-Q n 64
-q p 62
-Q y 64

R
-r 69

-R error 71
-R limit 71
-R min_weight 71
-R msg_id 71
-R msg_wrap 71
-R no_msg_id 71
-R no_msg_wrap 71
-R no_show_column 72
-R no_use_before_set 72
-R note 71
-R show_column 72
-R strict_errors 72
-R strict_warnings 72
-R suppress 72
-R use_before_set 72
-R warning 71
raw data file 109
read-only section 61
reference type in C++ 162
references
    external 12
    unresolved 65
register allocation
    feedback-directed 52
    per-procedure 51
register gp 55
register, storage class 141
reinterpret_cast 161
reserved keywords
    in C 133
    in C++ 159
right shift 143
runtime compatibility 125
runtime profile 62

S
-S 31
-s 69
scanner 10
severity code 80
shareable object 111, 116, 126
shared object 12, 66
sign of division remainder 142
sign propagation 142

size, command 112
size_t 141
software maintenance 126
source/error listing 73
standard C++ library 14
Static Analyser
    CIF information 75
static library 65, 116
static linking 12, 66, 111
statically linked library 12
string literals 146
Stroustrup, Bjarne 157
structure, C data type 134
structures 139
suffixes
    defaults for input file names 23
    defaults for output file names 27
    for input file names (user-specific) 28
    standard (for input file names) 76
    standard (for output file names) 76
switch statement 143
symbol, undefined 111
symbolic debugger 19, 59

T

-T add_prelink_files 48
-T all 47, 165
-T auto 47, 165
-T local 47, 165
-T max_iterations 47
-T none 47, 165
-T rem_prelink_libs 49
template instantiation 164
template options 47, 165
templates, C++ linkage 162
text segment 11, 112
threads
    C++ language support 186
    option 62
Tools.h++ 18

U

-U 39
-u 69

Ucode 97
    unoptimized 97
Ucode file 30, 51, 97
undefined symbol 111
union, C data type 134
unresolved references 65

V
-V 28
-v 72
virtual address space 12, 111
void
    C data type 133
volatile, type qualifier 141

W
-w 72
-W l 69
wchar_t 139
weak, #pragma directive 154

X
-X a 34
-X c 35
-X d 35
-X e 36
-X t 34
-X w 36

Y
-y 32
-Y F 28
-Y I 39
-Y P 70
-Y S 70

Z
-Z create_pch 39, 89
-Z pch 39, 87
-Z pch_dir 40
-Z pch_messages 40
-Z use_pch 39, 89

Contents

1	**Preface**	**1**
1.1	Documentation for CDS++ and additional references	2
1.2	Summary of contents	4
1.3	Notational conventions	5
2	**The C/C++ development system**	**7**
2.1	The C/C++ compiler	8
2.1.1	General characteristics	8
2.1.2	Structure of the compiler	10
2.2	The C++ libraries	14
2.2.1	The standard C++ library	14
2.2.2	The Cfront C++ library	17
2.2.3	The Tools.h++ library	18
2.3	The symbolic debugger DBX	19
3	**The cc, c89 and CC commands**	**21**
3.1	Command syntax and general rules	21
3.2	Description of options	26
3.2.1	General options	27
3.2.2	Options to select compilation phases	30
3.2.3	Options to select the language mode	34
3.2.4	Preprocessor options	37
3.2.5	Common frontend options in C and C++	41
3.2.6	C++-specific frontend options	44
	General C++ options	44
	Template options	47
3.2.7	Optimization options	51
	Options to select optimization levels	51
	Other optimization options	55
3.2.8	Options to control object generation	59
3.2.9	Link editor options	65
3.2.10	Options to control message output	71
3.2.11	Options to output listings and CIF information	73
3.3	Files	76
3.4	Predefined preprocessor names	78
3.5	Structure of compiler messages	80

4 Precompiled header files . . . 83
4.1 Basic aspects of PCH file handling . . . 84
4.2 Automatic PCH mode (-Z pch) . . . 87
4.3 Manual PCH mode (-Z create_pch, -Z use_pch) . . . 89
4.4 Other control possibilities . . . 90

5 Optimization . . . 91
5.1 Overview and general remarks on optimization . . . 91
General remarks on optimization . . . 93
5.2 Optimization phases in Ucode-based optimizations . . . 95
5.3 Ucode and dual object file . . . 97
5.4 Global optimizations . . . 99
5.5 Optimization with function inlining . . . 102
5.6 Feedback optimization . . . 103
5.6.1 Feedback optimization features . . . 104
5.6.2 Building the profiling executable . . . 106
5.6.3 Generating the profiling data . . . 109
5.6.4 Producing the optimized program . . . 110

6 The link editor . . . 111
6.1 Default settings and standard libraries . . . 114
6.2 Creating your own libraries . . . 116
6.2.1 Creating a statically linked library . . . 116
6.2.2 Creating a dynamically linked library . . . 116
6.3 Enhancing performance . . . 118
6.3.1 Minimizing the data segment . . . 118
6.3.2 Minimizing page faults . . . 120
6.4 Specifying directories . . . 121
6.5 Checking for runtime compatibility . . . 125
6.6 Linking library versions . . . 126

7 C language support of the compiler . . . 127
7.1 Overview of the C language modes . . . 128
7.2 Implementation-defined behavior based on the
ANSI/ISO C standard 136
7.3 Extensions to ANSI/ISO C . . . 146
7.4 Preprocessor directives . . . 149
7.4.1 Extensions . . . 149
#assert directive . . . 149
#unassert directive . . . 149
#ident directive . . . 150
#line directive (old format) . . . 150
7.4.2 #pragma directive . . . 150

8 C++ language support of the compiler 157
8.1 Overview of the C++ language modes 157
8.2 Implementation-defined behavior based on the
ANSI/ISO C++ standard 161
8.3 Template instantiation 164
8.3.1 Overview of instantiation modes 165
8.3.2 Automatic instantiation 166
8.3.3 Implicit inclusion 169
8.3.4 #pragma directives to control instantiation 171
8.3.5 Libraries and templates 173
8.4 Constructor and destructor calls for global and local static objects 178
8.4.1 Implementation in ANSI C++ modes 179
8.4.2 Implementation in Cfront C++ mode 181
8.4.3 Initialization and exception handling 182
8.5 Exception handling 183
8.6 Thread safety 186
8.6.1 Thread safety with DCE 186
8.6.2 Thread support in the C++ runtime system 186
8.7 Extensions to ANSI/ISO C++ 190
8.8 Variations in the Cfront C++ mode 192

9 Appendix: Overview of options (in alphabetic order) 199

Abbreviations 205

Related publications 207

Index 211

C/C++ Compiler V1.0
(Reliant UNIX)

User Guide

Target group
This manual addresses C and C++ programmers who work with the
C/C++ compiler of the CDS++ development system under Reliant UNIX.
Contents
- Overview of the C/C++ development system CDS++
- Compiling and linking of C and C++ programs with the commands cc, c89 and CC
- Programming notes and detailed information on: precompiled header files, optimization, binding, C and C++ language support of the compiler (implementation specific behaviour and extensions)

Edition: April 1997

File: cppe.pdf

BS2000 is registered trademarks of Siemens Nixdorf Informationssyteme AG.

Copyright © Siemens Nixdorf Informationssysteme AG, 1997.

All rights, including rights of translation, reproduction by printing, copying or similar methods, even of parts, are reserved.

Offenders will be liable for damages. All rights, including rights created by patent grant or registration of a utility model or design, are reserved.

Delivery subject to availability; right of technical modifications reserved.