



NCR UNIX SVR4 MP-RAS

High Performance C/C++ Compiler
Programming Guide
3.02.00

BD10-2613-E000
12/97

The product described in this book is a licensed product of NCR Corporation.

NCR is a registered trademark of NCR Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Limited.

General Notice: Other brands and product names used herein are for identification purposes only and may be trademarks of their respective companies.

It is the policy of NCR Corporation (NCR) to improve products as new technology, components, software, and firmware become available. NCR, therefore, reserves the right to change specifications without prior notice.

All features, functions, and operations described herein may not be marketed by NCR in all parts of the world. In some instances, photographs are of equipment prototypes. Therefore, before using this document, consult with your NCR representative or NCR office for information that is applicable and current.

To maintain the quality of our publications, we need your comments on the accuracy, clarity, organization, and value of this book.

Address correspondence to:

Systems Information
NCR Corporation
3325 Platt Springs Road
West Columbia, SC 29170

Copyright © 1997
By NCR Corporation
Dayton, Ohio U.S.A.
All Rights Reserved

Table of Contents

Preface

Who Should Read This Book.....	xiii
Releases Covered in This Book.....	xiii
What You Should Know	xiii
How To Use This Book.....	xiii
Conventions Used in This Book.....	xiv
Related Publications.....	xiv

Chapter 1

Features of High Performance C/C++

The C Language	1-2
The C++ Language.....	1-3
Portability	1-4
Safety and Efficiency.....	1-5
Other Features and Extensions	1-6
Optimizations	1-6

Chapter 2

Invoking the Compiler

The cc Command.....	2-2
Syntax.....	2-2
Example.....	2-3
Invoking the C Macro Preprocessor	2-4
Environment Identification.....	2-5
Predefined Macro Names	2-5
Predefined Asserts.....	2-5
Command-Line Options	2-6
General Options	2-6
Profiles	2-23
Example.....	2-23
Implementation.....	2-23
Defaults File	2-24
Format Strings for Output-File Names	2-26

Chapter 3

Compiler Pragmas

Using Pragmas	3-2
Pragma Reference	3-3
Pragmas	3-3
Customizing Your Program with Pragmas	3-13
Aligning Data	3-13
Using Initialization Pragmas	3-16
Aliasing External Names	3-18
Placing Comments in Object Code	3-20

Chapter 4

Compiler Toggles

What Is A Toggle?	4-2
Values and Descriptions of Compiler Toggles	4-3
Compiler Toggles	4-3

Chapter 5

Data Representation and Storage Mapping

Implementation Defined Behavior	5-2
Identifiers	5-2
Characters	5-3
Integers	5-4
Floating-Point Numbers	5-7
Arrays and Pointers	5-8
Using Keyword register to Place Objects in Registers	5-10
How the Compiler Maps Variables to Storage	5-10
Structures, Unions, Enumerations, and Bit Fields	5-12
Bit Fields	5-14
Accessing a Volatile Object	5-17
Maximum Number of Declarators	5-18
Maximum case Values in a switch Statement	5-18
Preprocessing Directives	5-18
Includable Source Files	5-18
Behavior on Encountering Pragma Directives	5-19
Compiler Limits	5-19
Storage Mapping for C++ Class Objects	5-21

Chapter 6

Run-Time Organization

Stack-Frame Layout	6-2
Parameters	6-3
Static link	6-3

Saving Registers.....	6-3
Direction Bit.....	6-4
Prologues and Epilogues	6-5
Function Results	6-8
Size of Result Returned In Registers.....	6-8
Exceptions.....	6-8

Chapter 7

Floating-Point Support

The 80387 Coprocessor.....	7-2
80387 In-line Instructions.....	7-3

Chapter 8

Externals

Interfacing to Other Languages	8-2
Undeclared Functions.....	8-3
Duplicate Function Declarations	8-4
Aliasing Pragmas.....	8-5
Alias	8-5
Example.....	8-6
Data Segmentation: the Data Pragma	8-7
Scope.....	8-8
Example.....	8-8

Chapter 9

Assembly Language Communication

Assembly Routines	9-2
Function Naming Conventions	9-3
Naming Conventions in C++.....	9-4
Calling Assembly Routines from High C/C++	9-4
Examples: Calling Assembly Routines from C or C++	9-5
Example 1.....	9-5
Example 2.....	9-5
Calling C or C++ from Assembly.....	9-7
Example.....	9-7
Data Communication.....	9-8
In-line Assembly Code	9-10

Chapter 10

Listings

Pragmas Page, Skip, Title.....	10-2
Format of Listings	10-3
Ruler.....	10-3

Include files	10-3
Example	10-5

Chapter 11

Compiling and Linking C++

Compiling and Linking C and C++ Modules	11-2
Do not use these identifiers in C	11-2
External Linkage between C and C++	11-3
Automatic “wrapping” of C include file definitions	11-4
Using the Built-In Inliner Versus the Stand-Alone Inliner	11-5
The Optional Stand-Alone Function Inliner	11-5
The Built-in C++ Inliner	11-6
Using the Name Unmanglers	11-6
Using Exception Handling	11-8
Interacting with Previously Compiled Classes	11-10
Standard Template Library (STL)	11-12
Generating Run-Time Type Information (RTTI)	11-13
Non-class Types	11-13
Class Types	11-13
Switching to New-Style Type Cast Notation	11-15
Limitations in Pointer-to-Member Comparison	11-16
Using Templates	11-17
The Single-Copy Problem	11-17
Using COMDAT Sections to Merge Global Instantiations	11-18
Instantiating Templates Manually	11-19
Overview	11-19
Preventing Duplicate Function Definitions	11-20
Selective Template Instantiation	11-23
Checking Template Usage	11-25
Problems with Matching const Parameters	11-26

Appendix A

Diagnostic Messages

File I/O Errors	A-2
System Errors	A-3
User Errors and Warnings	A-4
Lexical Error Messages	A-4
Syntactic Error Messages	A-5
Constraint Error and Warning Messages	A-5
Warning Levels	A-7
Error and Warning Messages	A-8
Changing the Message Format	A-22
Predefined Message Formats	A-22

Controlling Warning Messages	A-23
Controlling Groups of Warnings by Level	A-23
Controlling Individual Warnings by Number	A-23

Appendix B

Making Cross-References

Features of the Cross-Reference	B-2
Using the hcxref Command	B-3
Cross-Reference Pragmas	B-5
Supported Pragma_Names	B-5
Cross-Reference Format	B-7
Object-Name Cross-Reference	B-7
File Table	B-8
Module Name List	B-8
Annotated Source Listing	B-9
Distinction of File Names	B-10

Appendix C

Optimizing Program Performance

Introduction to Optimization	C-2
Specifying an Optimization Level	C-3
Debugging Optimized Code	C-4
Execution Speed Versus Code Size	C-5
Level 0 Optimizations	C-9
Global Register Allocation	C-9
Operator Strength Reductions	C-9
Local Common Subexpression Elimination	C-9
Constant Propagation	C-10
Constant Expression Folding	C-10
Eliminating Unreachable “Dead” Code	C-10
Live/Dead Analysis	C-10
Level 1 Optimizations	C-11
Preloading Arguments from Memory	C-11
Cross Jumping (Tail Merging)	C-11
Level 2 Optimizations	C-12
Iterative Local Common Subexpression Elimination	C-12
Iterative Live/Dead Analysis	C-12
Loop Memory Reference Elimination	C-12
Register Lifetime Analysis	C-12
Enhanced Register Allocation	C-13
Spill Analysis	C-13
Global Common Subexpression Elimination	C-13
Back-Substitution of Epilogue Code	C-13

Instruction Scheduling.....	C-14
Logical Predicates	C-14
Level 3 Optimizations.....	C-15
Instruction Scheduling.....	C-15
Peephole Optimizations.....	C-15
Improved Spill Analysis	C-16
Inlining	C-16
Enhanced Frame Removal.....	C-17
Resolve Branch Targets.....	C-18
Argument Passing in Registers	C-18
Additional Low level Scheduling	C-18
New Call Frame.....	C-18
Frame Pointer	C-18
Pointer-Based Variables	C-19
Level 5 Optimizations.....	C-20
Loop Strength Reduction.....	C-20
Level 6 Optimizations.....	C-21
Loop Unrolling.....	C-21
Interprocedural Optimizations	C-21
Level s Optimizations	C-22
Optimizing for Space.....	C-22
Other Level s Optimizations	C-22
Cache Optimizations.....	C-23
Processor-Specific Code Generation	C-24
Basic Block Optimizing.....	C-25
Utilities for Basic Block Optimization	C-25
Types of Basic Block Optimizations	C-25
When to Perform Basic Block Optimization	C-26
Options for Basic Block Optimization	C-26
Additional Ways to Optimize Performance	C-28
Tail Recursion	C-28
Function Prologue/Epilogue Optimizations.....	C-28
Function Calling Conventions	C-28
Constants in Code.....	C-29
Processor- Specific Instructions	C-29
Aligning Code and Data	C-30
Multi-Module Inlining.....	C-30
Expression Simplification.....	C-30
Compiler Intrinsics	C-31
Function Call Count Reporting.....	C-33
Structure Usage	C-35
Intermediate Compilation	C-37

Appendix D**Inline Function**

What Is An Inliner?	D-2
Options	D-3
Option Descriptions.....	D-3
Detailed Description of Options.....	D-4

Appendix E**Other Constructs and Utilities**

Intel 80486 Versus 80386	E-2
Intel Pentium Versus 80486	E-3
Intel Pentium Pro Versus 80486.....	E-4
Preprocessor Predicates.....	E-5
The Enhanced ASM Facility	E-6

Appendix F**High C Language Extensions**

Introduction to High C Language Extensions.....	F-2
X3J11 Extensions to C	F-3
Using \ as a Line Continuation Character.....	F-3
String-Constant Concatenation.....	F-3
Vertical Tab as a Delimiter	F-3
Suffixes u(U) and l(L) in Integer Constants	F-4
Suffixes f(F) and l(L) in Floating-Point Constants	F-4
Escape Sequences \a, \v, and \ddd in Strings and Characters	F-4
New Reserved Word signed	F-5
New type Denotation long double	F-5
Initialization of Automatic Aggregates With Static Expressions.....	F-5
Initialization of Structures With Structure-Valued Expressions	F-5
Function Prototypes.....	F-6
Multi-Character Constants.....	F-7
String Replacement (#) and Token-Pasting (##) in Macros	F-8
Trigraphs	F-11
High C Extensions Documented in the Manual.....	F-13
Underscores in numbers	F-13
Mix of declarations and statements	F-13
Case Ranges in the case Statement.....	F-14
Initialization of automatic aggregates with arbitrary expressions.....	F-14
New Reserved Word pragma.....	F-14
C++ Comment Style	F-14
C++ for Statement	F-15
Named Parameter Association.....	F-16
Syntax.....	F-16

Constraints.....	F-17
Nested Functions and Full-Function Variables.....	F-18
Nested Functions	F-18
Full-function Values	F-19
Non-Local Labels	F-21
Communication With Other Languages.....	F-23
Intrinsics	F-24
Descriptions.....	F-24
Packed and Unpacked Structures.....	F-29
__HighC__ Predefined Macro.....	F-30
Calling-Convention Attribute Specified with Function Declaration.....	F-31
Shiftjis Kanji Support	F-33
Direct I/O to Hardware Ports.....	F-34
In-Line Placement of Bytes of Code.....	F-35
ASM and Direct Register Assignment for Variables	F-36
Restrictions.....	F-38
short/long enum	F-40
ANSI Preprocessor Numbers.....	F-41
Switching to a New Base.....	F-41
Iterators.....	F-42
Some Uses for Iterators	F-43
Recursion and Code Clarity.....	F-45
Replacing Macros with Iterators.....	F-47
Syntax and Constraints	F-49
Semantics.....	F-51
Predefined Function Yield.....	F-53
Advantages and Disadvantages	F-53
Conclusion.....	F-55
Prototypes.....	F-57

Appendix G

Loop Unrolling

Introduction to Loop Unrolling	G-2
Selecting Loops for Unrolling	G-3
How Loops are Unrolled	G-4

Appendix H

Structure Report and Call Graph Facility

Getting and Using a Call Graph.....	H-2
Site Specific Inlining Using a Call Graph.....	H-2
Function Reordering at Link Edit Time.....	H-3
Merging Call Graph Files	H-4
Structure Usage Reporting.....	H-6

Format of a Structure Usage Line.....	H-6
Six profcnt Report Types	H-7
getFieldUse Warning Message.....	H-8

Appendix I

Intermediate Compilation

Building Intermediate Object Code Files	I-3
Controlling Inline Substitution in Intermediate Object Code Files.....	I-4
Site Specific Inline Substitution	I-5
Building Libraries From Intermediate Object Files.....	I-7
Building Normal and Intermediate Object Files in the Same Directory	I-8
Environment Variables With the Intermediate Compilation Facility.....	I-9
TMPPREFIX and TMPDIR.....	I-9
ILOPT_ARGS.....	I-10
Output From a Command Line Using -Hih and -Hit=64	I-11

Preface

Who Should Read This Book

This book is for the person who uses the High Performance C/C++ Compiler (also referred to as High C) as hosted on the UNIX[®] operating system for the 80386, 80486, Pentium, and Pentium Pro family of processors.

Releases Covered in This Book

This book applies to Release 3.02 of the NCR UNIX SVR4 SCDE Package.

What You Should Know

You should be familiar with the UNIX operating system and with the C or C++ programming language.

How To Use This Book

This guide contains all system-specific information necessary for using the compiler effectively. Readers new to the product should scan the Table of Contents for an overview of the guide.

This guide describes the following:

- how to compile, link, and run a program
- how to use compiler controls: options, profiles, pragmas, and toggles
- machine specifics, such as storage mapping and run-time organization
- defaults and limits

- communication with programs written in other languages
- listings
- error messages
- language extensions

Conventions Used in This Book

The following conventions are used throughout this book:

- Names of packages, commands, and utilities are printed in **bold** type.
- Directory names, file names, and variables are printed in *italic* type.
- Screen output and source code is printed in `courier` type.
- Numbered lists denote things that must be performed in a specific order. Bulleted lists do not denote specific order.
- Compiler pragmas are printed in **bold** type.
- Compiler toggles are printed in *italic* type.
- Compiler command-line options are printed in **bold** type.
- A reference to a chapter section is enclosed in quotes, such as, Chapter 11, “The Single-Copy Problem”.

Related Publications

Refer to the following for additional information:

- *NCR UNIX SVR4 MP-RAS Reference Manual*
- *NCR System 3000 Standard C Development Environment (SCDE) Release Definitions*
- *Intel i386/i486/Pentium Microprocessor Programmer’s Reference Manual*
- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup
- *ANSI/ISO 9899-1990 for Programming Language C*
- *ANSI/IEEE Standard 754/1985 for Floating Point Implementation*

Features of High Performance C/C++

This chapter gives an overview of the C and C++ languages and covers the features of High Performance C/C++. The following topics are included in this chapter:

- The C Language
- The C++ Language
- Portability
- Safety and Efficiency
- Other Features and Extensions

The C Language

C is a mixed-level systems language designed by Dennis Ritchie at AT&T's Bell Laboratories. The C language grew in popularity because of its use in implementing the UNIX operating system, its elegant (and deceptive) simplicity, and its close-to-the-machine features. Many software developers now use C for applications, as well as for systems software.

Later implementations of C were extended to add enumeration types and a few other features. More recently many extensions have been proposed to make C a safer language while still being consistent with the philosophy of the original language. Today a core language has been standardized by the American National Standards Institute (ANSI).

High C includes ANSI Standard C and also provides extensions that were carefully designed to be consistent with the philosophy of C. Some of the best features of such other languages as Pascal, MIT's CLU, and Ada have been incorporated as extensions. Incompatibilities were minimized by introducing a minimum of new keywords and by retaining the original syntax. Yet the extensions are such that they will be flagged by any Standard-conforming compiler.

It also supports existing Pre-ANSI C programs by offering a transition compile mode. This permits most Pre-ANSI C programs to compile, with warning messages for constructs that differ in interpretation if they are compiled in ANSI Standard C mode.

The C++ Language

The C language was extended in 1980 to add several significant new features, intended to make the language more robust for large programming projects. Bjourne Stroustrup added the concept of classes, the key new data type abstracter, and several other features. After many refinements, the C++ language was defined in 1984.

Over time, several additional facilities, such as virtual functions C, operator overloading, multiple inheritance, type safe linkage, templates, and exception handling were added. *The Annotated C++ Reference Manual* by Margaret Ellis and Bjourne Stroustrup gave a definitive basis for the language in 1990, and move to the ANSI X3J16 Standardization process.

High Performance C++ includes all of the non-experiential portions of *The Annotated C++ Reference Manual*, and implements several additional features from the ANSI X3J16 drafts, including templates, namespaces, exception handling, and runtime type information. All language constructs and library APIs supported by C front 3.0 implementations are present.

Portability

Standard C programs can be compiled with an ANSI option that turns off the extensions and reduces the language to the Standard core. Alternatively, such programs can be gradually upgraded by not choosing the ANSI option and using the extensions as required.

Objects are backward portable to earlier versions of SCDE Release 2, and are portable to other iABI-compliant systems.

Portability to other implementations of C++, including the ncc compiler, is at the source level. Binary compatibility should not be expected. This is also true for users of the **ncc** package C/C++ compiler; all CH libraries and object files should be regenerated with the new compiler.

Safety and Efficiency

While the close-to-the-machine features of C are available, High C also supplies the new strong type checking specified in ANSI C. In addition, the compiler provides many checking features usually available only in a separate lint program. Thus, one gets both efficiency and reliability.

Other Features and Extensions

Other important features and extensions include:

- Code generation optimized for the 386, 486, Pentium, or Pentium Pro processor
- Function inlining (also known as function integration)
- Compiler controls and options, including one for strict Standard checking
- “Iterators” for data abstraction and driving for loops, as in MIT’s CLU language
- Nested functions complete with up-level references, as in Pascal
- Nested functions passable as parameters to other functions, as in Pascal
- Lint-like diagnostics and a toggle to turn them off
- Compatibility with the AT&T portable C compiler (PCC) by way of the *PCC* toggle, **-Hpcc** command-line option, and the **-Xt** option
- Intrinsic functions, such as *_abs*, *_min*, and *_max*, for efficiency

Optimizations

Optimizations, some of which are usually found only in mainframe and supercomputer compilers, including:

- Global flow analysis and removal of loop invariant expressions
- Live/dead analysis
- Global common subexpression elimination
- Global register allocation
- Dead code elimination
- Constant folding
- Constant and copy propagation
- Numerous strength reductions
- Automatic mapping of variables to registers
- Cross jumping (tail merging)

- Profile-driven program restructuring for efficiency, including:
 - Basic block cloning and restructuring to minimize branch impact
 - Profile-driven register allocation
 - Superblock formation
 - Profile-driven switch statement optimization
- Scheduling for pipeline and superscalar throughput
- Loop unrolling and induction variable elimination
- Wide compiling to optimize across module boundaries
- Static branch prediction
- Interprocedural optimizations
- Optimizations targeted for the I386, I486, Pentium, and Pentium Pro processes

Invoking the Compiler

The following topics are included in this chapter:

- The cc Command
- Invoking the C Macro Preprocessor
- Environment Identification
- Command-Line Options
- Profiles
- Defaults File

The cc Command

The **cc** command invokes the High C compiler, which translates C programs into executable load modules, relocatable binary object modules, or assembly-language source files.

Syntax

The syntax of the command is:

cc [options]... files...

Any number of options and one or more files can be specified. Each option specified in the command applies to all the specified files for which it makes sense, except as noted below.

Several types of file names are allowed. A file name ending with **.c** or **.i** is taken to be a C source module. It is compiled and an object module is produced with the same name as the source except with **.o** substituted for **.c** or **.i**. The **.o** file is normally deleted when a single-module C program is compiled and linked.

A file ending with **.C**, **.cc**, **.cpp**, or **.ii** is taken to be a C++ source module, with the same conventions as C source modules.

The command line option **-Hcppext** can also be used to change the C++ file name extensions to match a specific build environment.

A file name ending with **.s** is taken to be an assembly source module and is assembled, producing a **.o** file. Any other file specification is assumed to be an object module or archive library to be passed to the linker.

All **.o** files are placed in the current working directory.

In general, the linker is invoked if no compilation errors were detected and the **-c** or **-s** options were not specified. The resultant load module is named **a.out** unless specified otherwise with the **-o** option (described below). Any argument beginning with a dash (**-**) is taken as an option specification.

Example

The following command compiles the program in file *sort.c*, links it, and generates a load module named *sort*:

```
cc -o sort sort.c
```

Invoking the C Macro Preprocessor

The High C compiler has an integrated internal macro preprocessor. The preprocessor conforms to the proposed ANSI C Standard. However the external C macro preprocessor on most UNIX operating systems does not conform to the proposed Standard in some ways.

Typically, an external preprocessor is available, and the **-Hcpp/-Hnocpp** options are provided to use it.

- **-Hcpp** causes the external ANSI C preprocessor to be invoked on the source file sending the output to a temporary file, which then serves as input to the compiler.
- **-Hnocpp** suppresses this action. The preprocessor integrated within the C compiler is used. This is the default.

Environment Identification

Predefined Macro Names

For programs to identify the environment in which they are running, specific macro names are predefined.

- `_I386`, `i386`, `unix`, `_ATT4`, and `_nst` are always predefined for **-Xt** and **-Xa**, and **C++** compile modes.
- `__HIGHC__` is predefined for all compile modes and C++.
- `__STDC__` is always predefined.
 - For **-Hansi** or **-Xc** compiles, it is defined to be **1**.
 - For **-Xt** (default) or **-Xa** compiles, it is defined to be **0**.
 - For C++ compiles, it is defined to be **0**
- `__cplusplus` and `__CPLUSPLUS__` are predefined for C++ compiles.

Predefined Asserts

The following asserts are always defined:

- `#assert system (unix)`
- `#assert cpu (i386)`
- `#assert machine (i386)`
- `#assert machine (_nst)`

See the “Preprocessor Predicates” section in Appendix E, “Other Constructs and Utilities” for more information on asserts and how you can use them.

Command-Line Options

Each command-line option is described below. Any option that is not recognized by **cc** is assumed to be a linker option and is passed on to the linker. Options applicable only to High C are prefixed with an **H** to avoid conflicts with existing or future PCC options.

General Options

The following are the command-line options. Note that code specifically generated for the 80386, 80486, or Pentium CPU executes properly on systems with any of these CPUs, but it executes more efficiently if it is tuned for the target system's CPU.

- **-#**: Display the pathnames and arguments for each utility that executes as part of the compile process.
- **-386**: Tune code generation for a 80386 processor (High Performance C only).
- **-486**: Tune code generation for a 80486 processor. This is the default behavior for SCDE 2.00 and 2.01 (High Performance C only).
- **-586**: Tune code generation for a Pentium processor. This is the default behavior for SCDE 2.02 (High Performance C only).
- **-686**: Tune code generation for a Pentium Pro processor, without using Pentium Pro unique instructions sequences. This permits the code to run on Pentium or 80486 processors. This is the default behavior for SCDE 3.01 and later versions (High Performance C only).
- **-A name (token)**: Associates name as a predicate with the specified token as if by a **#assert** preprocessing directive. See the section "Preprocessor Predicates" in Appendix E, "Other Constructs and Utilities". The following predicates are predefined:
 - system (unix)
 - cpu (i386)
 - machine (i386)
 - machine (_nst)

- **-A-:** Causes all predefined macros (other than those that begin with “__”) and predefined assertions to be forgotten.
- **-C:** Causes the preprocessing phase to pass along all comments other than those on preprocessing directive lines.
- **-c:** Suppresses the invocation of the linker, and forces an object file to be produced even if only one module is compiled.
- **-Dname, -Dname=def:** Defines the name **name** to the preprocessor as if by #define. If no **def** is given, the value of name is defined to be 1.
- **-E:** Specifies that the C macro preprocessor is to be invoked and no compilation done. The preprocessor output is sent to standard output.
- **-F=<pathname>:** Specifies a file with specific inlining information to be passed to the inliner. This permits selective inlining of functions at call sites, and can be used to significantly improve performance of call intensive programs.

See Appendix D, “Inliner Function” for more information on the inliner and use of this option.

- **-fdouble:** Specifies that all floating-point computations are to be done in no less than double precision. Operands of type float are widened appropriately. This makes floating-point computations more accurate but can slow performance. (This is the default for the **-Xt** compile mode.)
- **-fsingle:** Specifies that single precision arithmetic is to be used in floating-point computations involving only float expressions. This is the default for the **-Xa**, **-Xc**, and **-Hansi** compile modes.
- **-fstrict:** Directs the compiler to generate strictly ANSI C conforming code, at the expense of run-time speed. It will truncate the size of values on the 387 floating point stack to the proper size according to the ANSI C standard. This is the default for the **-Xc** and **-Hansi** compile modes.
- **-g:** Causes the compiler to emit debug information. **-g** turns off many of the optimizations that the compiler otherwise performs. The **-ilobj** option is mutually exclusive with the **-g** option. Do not use the **-ilobj** option with **-g**.
- **-H:** Causes the compiler to display the name of each file included during compilation on standard error output, one name per line.

- **-H except:** Specifies that all classes in this C++ source module need to be compiled exception aware. The C++ runtime library is generated with this option, and so should be safe for any exception handling programs using classes from this library.
- **-Hanno:** Used in conjunction with the **-S** option (explained below). Specifies that the generated `.s` file is to be annotated with lines from the source file.
- **-Hansi:** Causes the compiler to accept only programs conforming to the ANSI C Standard. This mode is stricter than the UNIX **-Xc** compile mode.
- **-Hasm:** Causes the compiler to produce a (pseudo-) assembly listing of the generated code on standard output, by initializing the *Asm* toggle to On. The assembly listing is annotated with lines from the main source file, but not with lines from any included files. These lines appear as comments immediately preceding the corresponding assembly instructions. If the **-S** option (described below) is also specified, the generated `.s` file is annotated with lines from the source file, and no listing is written on standard output; that is, it has the same effect as **-Hanno**.
- **-Hbbarc=filename:** Optimizes using the basic block dynamic profile data counts that were generated by an executable module compiled with the **-qb** option. See "Basic Block Optimizing" in Appendix C, "Optimizing Program Performance" for more information.
- **-Hc_wrap={xxx|^yyy}:** Interpret header files included in C++ files as C, not C++ (C++ only)

Directs the compiler to treat files included in C++ files as C header files rather than as C++ header files. If an include file has `xxx` anywhere in its name (for **-Hc_wrap=xxx**), or begins with `yyy` (for **-Hc_wrap=^yyy**), the include file is preceded by file `c_wrap1.h` and followed by file `c_wrapr.h` (included in the SCDE distribution). These two files cause the include file to be interpreted as a C header file, not a C++ header file.

You can use this option to include old C header files with C++ code. A common usage would be:

```
-Hc_wrap=/home/joe/cinclude
```

You might have other include files that you want to use with C++ programs; you can specify **-Hc_wrap** any number of times. Option **-Hc_wrap** turns Off toggle *Empty_is_void* in file *c_wrap.h*.

- **-Hcomdat**: Manually enables COMDAT solution for template instantiation (C++ only)

The COMDAT solution is the default solution for template instantiation and virtual function table generation when you use the SCDE linker. When COMDAT support is enabled, these toggles are turned On:

- *Inline_common*
- *RTTI_common*
- *Template_common*
- *Vtable_common*

To disable COMDAT support (for example, if you are using a linker on another vendor's system), specify option **-Hnocomdat**.

For additional information, see the following:

- option **-Hnocomdat**
- Chapter 11, “Compiling and Linking C++” (“The Single-Copy Problem”)
- Chapter 11, “Compiling and Linking C++” (“Instantiating Templates Manually”)
- **-Hcplus**: Include C++ header files at compile time and link C++ run-time library at link time. The compiler uses this option by default. See also **-Hnocplus**, which has the opposite effect. For more on compiling C++ programs, see Chapter 11, “Compiling and Linking C++.”
- **-Hcpp**: Specifies that the external ANSI C macro preprocessor (*/usr/ccs/lib/acpp*) is to be used, rather than the internal preprocessor.
 - **-Hcpplvl=n** - Specify the level of C++ compilation (C++ only). Specifies the level (n) of C++ compilation to be used by the compiler, where the value of n is 1, 2, or 3. When you specify level 1, the compiler applies a limited number of C++ rules; at level 3, all C++ rules are rigidly enforced. The default level is 3.
 - **Level 1 C++:**

- Level 1 is "weak" C++. All features of C++ are available, through possibly alternative syntax, but certain aspects of C++ that change the semantics of ANSI Standard C programs are not enabled.
- The following characteristics hold at level 1:
 - The type of an enumeration literal is `int`
 - The type of a character is an integer type rather than type `char`
 - The storage class of constant objects is not `static` by default
 - Functions are not by default overloadable
 - An `int` can be converted into an enum type
 - A struct or union does not generate a local scope, but a class does
 - Anonymous unions behave as in ANSI Standard C -; that is, they do (almost) nothing.
 - Class names do not become typedef names
 - In a function declaration, an argument specification of `()` is not interpreted as `(void)` unless the function appears within a class.
 - The following words are not reserved; access to these C++ features must be through alternative syntax:
`catch, class, delete, friend, inline, namespace, new, operator, private, protected, public, template, this, throw, using, virtual`
- **Level 2 C++:**
 - Level 2 means that all of C++ is implemented, but certain prohibitions in the C++ language are not enforced:
 - The C++ language requires that a reference can be bound to either (a) something of the same type (or of a derived type), or (b) something of a compatible type, but the reference must be to a `const`. The restriction that the reference is to a `const` is not enforced at this level, due to prior practice.
 - Converting an `int` into enum generates only a warning.

- Calling an undeclared function is allowed. The compiler assumes an undeclared function is an externally available C function.
- Initializing an array of *n* chars with a string of length *n* is allowed.
- **Level 3 C++:**
 - Level 3 is full C++, plus enforcement of the restrictions relaxed at level 2.
 - Legacy C++ programs that compiled with early versions of the **cfront** or **ncc** compiler may require the **-Hcpplvl=2** option to compile properly.
- **-Hcppext=ext1, ...:** Specifies the C++ source file extensions. The compiler will recognize files with these extensions as C++ source files and will compile them in C++ mode.

For example:

```
cc -Hcppext=cplus x.c y.cplus
```

will compile source module 'x.c' as a C source file and 'y.cplus' as a C++ source file.

By default, the compiler recognizes the source file extensions of '.C', '.cc', '.cpp', and '.ii' as C++ source files.

- **-Hefile={ @ [E] | filename }:** Redirect output sent to standard error either to standard output or to a file. Tells the compiler to redirect output sent to standard error, either to standard output or to a file. This option is particularly useful on systems that do not have a convenient way to redirect standard error. The syntax of the **-Hefile** option is:

-Hefile=@	Direct output to standard output
-Hefile=@E	Direct output to standard error
-Hefile=filename	Direct output to file <i>filename</i>

Option **-Hefile** can take a format string in place of a specific *filename*. See the section entitled "Format Strings for Output-File Names" for details.

- **-Hefmat=format:** Set format of error messages. Directs the compiler to format error messages according to the variable *format*, where *format* is a quoted string containing a sequence of characters with special meanings.

For more information, see Appendix A, “Diagnostic Messages”.

- **-Hgac=string:** Sets the global aliasing convention used for determining the mapping of internal names to external names.

For more information, see the definition for pragma

Global_aliasing_convention in the “Pragma Reference” section of Chapter 3, “Compiler Pragmas”.

- **-Hi, -Hia, -Hic=n, -Hih, -Hin, -Hir, -Hit=n, -Hix, -Hiz, -Hiw, -His=n, -HiC=n, -HiS=n, -HiT=n:** Inliner options. See Appendix D, “Inliner Function,” for descriptions.
- **-Hinlsize=size:** Specify the maximum size of inlined functions (C++ only). Specifies the maximum size of functions to be inlined by the C++ compiler’s built-in inliner. The greater the value of the variable **size**, the greater the size of functions that can be inlined.

The default value for **size** is 100, which is typically about 15 statements. Specifically, the value of **size** is the number of expression tree nodes required to represent the function.

This option applies to the front end inliner and not the standalone function inliner.

- **-Hkanji:** Specifies that the compiler is to use the Kanji scan tables. This enables Kanji support, which is otherwise disabled.
- **-Hlines=n:** Causes a page eject to occur after every *n* lines written to standard output. The default of 60 is appropriate for most 6-lines-per-inch printers, which allow a maximum of 66 lines per page for 11-inch paper. The setting of **-Hlines** is intended to allow some blank space at page boundaries. For 8-lines-per-inch (88 lines per page) printers, **-Hlines** should be set to 80 or 82. This option is used in conjunction with the **-Hlist** and **-Hasm** options. If *n* is 0, no page ejects are emitted.

- **-Hlist:** Causes the compiler to generate a source listing on standard output by initializing the List toggle to On. See Chapter 4, “Compiler Toggles.”
- **-Hmake(=macro):** Directs the compiler to produce a “makefile” on standard output. No object module is generated nor is the linker invoked. This produces a makefile “dependency entry” for the object file corresponding to each source file. The object file is made dependent on the source file and any files included in its compilation. If a macro is defined, it is appended to the front of each dependency entry. This allows relative paths for make scripts.
- **-Hmakeof=m_f_name:** Write a list of makefile dependencies to the named file. Directs the compiler to generate dependencies that can be used to create a makefile, and to redirect the list of such files to the named file. The list contains the names of files included in the source file, as well as the name of the source file itself. Option **-Hmakeof** can take a format string in place of a specific **m_f_name**.

For more information, see:

- options **-Hmake**, **-Hrel**, and **-Hmake**
- toggle *Normalize_file_names*
- **-Hnocomdat:** Disable the COMDAT solution for template instantiation (C++ only).

The COMDAT solution is the default solution for template instantiation and virtual function table generation when you use the SCDE linker. If you need to link with a linker on another vendor's system, you must specify option **-Hnocomdat** to disable COMDAT support, and adopt one of the manual instantiation techniques described in Chapter 11, “Compiling and Linking C++” (“Instantiating Templates Manually”). When you specify option **-Hnocomdat**, these toggles are turned off:

- *Inline_common*
- *RTTI_common*
- *Template_common*
- *Vtable_common*

For more information about COMDAT support, see Chapter 11, “Compiling and Linking C++” (“Using COMDAT Sections to Merge Global Instantiations”).

- **-Hnocplus:** Do not include C++ header files or link in the C++ run-time library.

Instructs the compiler to exclude C++ header files and the C++ run-time library when a program is compiled and linked.

CAUTION: Do not use option **-Hnocplus** with C++ programs, because C++ programs must be linked with the C++ run-time library. However, you can slightly increase the efficiency of your compilations by using this option to compile programs written in C.

For more information about compiling C++ programs, see Chapter 11, “Compiling and Linking C++” (“C++-Specific Issues”). See also option **-Hcplus**, which has the opposite effect. The compiler uses option **-Hcplus** by default.

- **-Hnocpp:** Specifies that the internal C macro preprocessor is to be used. See the section entitled “Invoking the C Macro Preprocessor” in this chapter for details. This is the default.
- **-Hnoobject:** Do not generate object code (only check for compilation errors).
- **-Hnopro, -Hpro=file:** Specifies a “profile,” a file that is read by the compiler before reading the source file. If this option is omitted, the default hc.pro applies (see the “Profiles” section of this chapter for details). **-Hnopro** suppresses reading the profile.
- **-Hon=toggle1, toggle2, ...togglen, -Hoff=toggle1, toggle2, ...togglen:** Turns a toggle On or Off. See Chapter 4, “Compiler Toggles,” for additional information.
- **-Hpcc :** Specifies that the compiler is to run in “PCC” mode. In this mode, the compiler relaxes enough of the ANSI extensions to more or less emulate the Portable C Compiler. This permits old C programs that would not ordinarily compile to compile with little (if any) modification. Specifically, **-Hpcc** does the following:
 - Turns on the PCC toggle (see Chapter 4, “Compiler Toggles.”)
 - Unreserves the following keywords: signed, volatile, and pragma
 - Turns on the *Long_enums* toggle so that enum types are mapped to full words, as is the PCC convention.

Use of the **-Xt** compiler option is recommended to get this functionality.

- **-Hpragma=pragma_name:** Invokes `pragma_name` just as if it were invoked in the source file. See Chapter 3, “Compiler Pragmas” for more information on pragmas.
- **-Hrel:** When dependency entries are produced with the **-Hmake** or the **-Hmakeof** option, the **-Hrel** option directs the compiler not to add absolute include files (those included with `< >`).
- **-Hsrc=file:** Specifies that debug line numbers should be generated for ‘file’ instead of the compile source filename. This is useful for language translators that produce C source files as output with debugging ‘#line’ directives. This permits line breakpoints to be set in the original source module.
- **-Hsuffix=suffix:** Directs the compiler to produce an object file ending in `.suffix`.
- **-Htmpprefix=temp_dir:** Sets directory **temp_dir** for temporary compile process files. This option supersedes the setting of the `TMPPREFIX` shell variable.
- **-Hmake[=macro]:** Directs the compiler to produce a `.u` file that contains a list of include files contained in the source file, and the name of the source file itself. These entries are the dependent components for the source file. If the optional parameter `macro` is defined, it is prefixed to each entry in the `.u` file. The `.u` file is created in the same directory where the source file is located.
- **-Hunroll=fff,sss:** Causes the compiler to replace a loop with the code that makes up the body of the loop. The parameter `fff` is the maximum times that the loop will be unrolled, and `sss` is the maximum size of a loop that can be unrolled. Loop unrolling reduces the loop overhead but increases the size of the object generated. See Appendix G, “Loop Unrolling”, for a detailed description (SCDE Release 2.01 and later).
- **-Hwide:** Specifies multi-module inlining when **-Hwide** is specified with **-Hiw**. See Appendix D, “Inliner Function” for more information. The modules being compiled are treated as a collection of modules to be compiled, instead of individually. This permits cross-module inlining of functions (SCDE Release 2.01 and later).

- **-Idir**: Specifies an alternate directory to be searched to locate an include file. This option can be specified several times to indicate several directories to be searched. If a particular file is not located after searching the specified directories, one or more standard directories are searched. The directories are searched in the order that they are specified. See Chapter 3, “Compiler Pragmas” for more information.
- **-ilobj**: Creates intermediate objects instead of relocatable object modules. This incurs the opportunities for cross module optimizations. See Appendix C, “Intermediate Optimizations.”
- **-K [options]**: Enable a specific compiler code generation strategy. Most of these options have been provided to give UnixWare 2 compatibility.
 - **-K abiplus** directs the compilation system to use a version of the C library with functions added to the shared library for compatibility with the **ABI+** standard. Note that applications that use this option to link may not be portable to earlier versions of the operating system without updating the shared C library, as the additional shared library entry points will be missing in the earlier release.
 - **-K blended** is equivalent to **-686**
 - **-K catchnull** segment fault on NULL pointer reads. By default a NULL pointer memory read will return zero. With this option a segmentation fault will be generated instead. This is useful in finding potential NULL pointer dereference instances in programs that are usually latent bugs or are at best non portable.
 - **-K cplusplus** force in C++ constructor/destructor support. This is intended for shared libraries with C++ code that require constructors be run for objects, but do not want to link with the full C++ support library. This brings in the support to invoke constructors at library initialization time, instead of the default stub modules that take no action.
 - **-K frame** generates code that uses a local call frame saved in register **ebp** for all functions.

- **-K host** (default) enables inlining of ANSI C library and math library functions, with the assumption that the ANSI C names and functionality are expected. With this option enabled the functions will be expanded inline and no library call will be made. With this option enabled some of the math library error handling and calls to `matherr` will be disabled.
- **-K i386** is equivalent to **-386**
- **-K i486** is equivalent to **-486**
- **-K_ieee** forces intermediate floating point results to be stored in temporary storage, truncated to the size of the types in an expression. This has the same effect as specifying the **-fsoic** option. This is required for full compliance with the ANSI/ISO C standard, and is enabled when the **-Xc** or **-Hansi** options are specified.
- **-K inline** enables inlining to the default optimization level. This is equivalent to specifying the **-Hit=-24** option to the inliner.
- **-K minabi** directs the compilation system to use a version of the C library that minimizes dynamic linking, without changing the application's ABI conformance (or nonconformance, as the case may be). Applications that use the Network Services Library or the X library may not use **-K minabi**.
- **-K no_host** disables ANSI C library function inlining and so calls the functions from the C library or application.
- **-K no_ieee** disables intermediate result precision truncation.
By default the compiler saves temporary variables on the floating point stack in full long double precision, regardless of the types of the variables in an expression. Enabling truncation will have a negative impact on the performance of floating point intensive programs.
- **-K noframe** generates code without using a local call frame saved in register **ebp**. This is the default behavior and should generate the most efficient code.
- **-K nonlinear** disables inlining. This is equivalent to specifying the **-Hit=O** option to the inliner.
- **-K P5** is equivalent to **-586**
- **-K P6** is equivalent to **-686**

- **-K PIC** causes position-independent code (PIC) to be generated.
 - **-K pentium** is equivalent to -586.
 - **-K schar** (default) makes plain character types signed. Note that the link time libraries provided with the SCDE package all are compiled in this mode, so character handling by the libraries will always use signed character handling.
 - **-K uchar** makes plain character types unsigned.
 - **-K xpg42** link with the C library that provides XPG 4.2 behavior. By default, the C library provides legacy MP-RAS behavior (SCDE 3.03 with MP-RAS 3.02.00 or later).
 - **-O, -On:** Sets the optimization level to n. If n is not specified, the optimization level is set to 3. See the section entitled “Specifying an Optimization Level” in Appendix C, “Optimizing Program Performance”.
 - **-o output:** This option is passed on to the linker and names the final executable output file output. When this option is used, any existing a.out file is left undisturbed.
- Note:** Whitespace is required after the **-o**.
- **-P:** Runs only the preprocessor on the named source files and leaves the result in corresponding files suffixed with .i. If **-Hcpp** is used in conjunction with **-P**, the **-P** option is passed to **cpp(1)**.
 - **-p, -qp:** Produces code that counts the number of times each function is called during execution. If the linker is invoked, the standard start-up function is replaced with one that automatically calls **monitor(3)** at the start and writes a *mon.out* file. An execution profile can then be generated by using **prof(1)**.
 - **-pg:** Arranges for the compiler to produce code that counts the number of times each function is called by each of its callers. If link editing takes place and the **-dn** option is used, profiled version of *libc.a* and *libm.a* (with the **-lm** option) are linked. At normal termination, the linked executable writes a *gmon.out* file. An execution profile with call graph call counts and timing can then be generated by the **gprof(1)** command.

Setting environment variable `GPROF_USE_PID` to any value prior to program execution alters the name `gmon.out` to `gmon.PID`, where `PID` is the process ID of the program. Setting environment `GMON_OUT_NAME` to the path name (absolute or relative) of a file prior to program execution directs the **gprof** exit code to write the profile data to the named file. If both environment variables are set, `GPROF_USE_PID` takes precedence over `GMON_OUT_NAME`.

If no link editing occurs, the **-p** and **-pg** options are equivalent. It is only at link edit time that the decision whether to use **prof** or **gprof** style profiling must be made. If neither profiling option is used at link edit time, no profiling occurs.

The **-pg** option overrides all occurrences of the **-g** option.

- **-Qc**: Enables the generation of identification information by each tool invoked during a compile into the output files. `c` can be either `y` or `n`. This can be useful for tracking the version of each compile process utility. The information is added to the `.comment` section in the object module and can be displayed by using the **mcs(1)** command. Giving `n` for `c` suppresses this information; it is enabled by default.
- **-qc**: Enables profiling options. `c` can be `l`, `p`, `b`, or `s`.
 - **-ql** causes the invocation of the basic block analyzer and arranges for the production of code that counts the number of times each source line is executed. A listing of these counts can be generated by use of `lprof`.
 - **-qp** is a synonym for **-p** (see the **-p** option above).

The **-qb** adds basic block profiling code and links in a library to manage the accumulation of the basic block counts and the output of a count file at the end of execution.

The **-qs** specifies basic block profiling code and links in a library to manage the accumulation of the basic block counts and the output of a count file at the end of execution.

See **profcnt(1)** for more information on how to retrieve profile information about an executable compiled with either **-qb** or **-qs**. See the section “Basic Block Optimizing” in Appendix C, “Optimizing Program Performance” for more information on using and optimizing with **-qb** and **-qs**.

The **-ilobj** option is mutually exclusive with the **-ql** option. The user must remove one of them.

- **-S**: Produces an assembly source file instead of an object file for each source file. The assembly source is written to a file with the same name as the C source, with `.c` replaced by `.s`. The file is always placed in the current working directory. No object file is written, nor is the linker invoked.

If **-Hanno** is also specified, the assembly file is annotated with lines from the source file as comments.

- **-Uname**: Removes any initial definition of macro name. See **-D** above.
- **-V**: Prints version information for each compile process tool on standard error.
- **-v**: This option causes the compiler to perform more complete semantic checking by setting the warning level to 4. See the section entitled “Warning Levels” in Appendix A, “Diagnostic Messages”.
- **-W tool,arg1[,arg2...]**: Causes the compiler to hand off the arguments `argn` to `tool`. Each argument must be separated from the next with only a comma. See section `cc(1)` in the Reference Manual.
- **-wn**: Determines which level of warnings the compiler issues (see the section entitled “Warning Levels” in Appendix A, “Diagnostic Messages”). `n` may not be less than 0, nor greater than 4. Each warning has an implicit level based on how important it might be to the programmer. The higher the level, the less important it is considered to be. **-wn** suppresses all warnings at level `n` and higher. Thus, **-w0** suppresses all warning messages; **-w4** enables all of them. The default is **-w3**.

If **-w** is specified without a level number, level 0 is assumed; that is, all warning messages are suppressed.

Note: **-wn** is equivalent to:
-Hpragma=“Warning_Level (n)”.

- **-Xa, -Xc, -Xt**: Specifies the degree of conformance to the ANSI standard by designating the mode in which the compiler is to run.

- **-Xa:** The ANSI mode is similar to the transition mode (**-Xt**) except the compiler decides in favor of ANSI C when encountering a construct with differing interpretations. Therefore, the *Use_UP_rules* toggle is turned Off.

Note: This mode is misnamed; it does not enforce strict ANSI conformance and should not be confused with the **-Hansi** option.

- **-Xc:** The conformance mode enforces most ANSI-restrictions and accepts all High C extensions. Standard header files are ANSI-compliant and only ANSI-defined identifiers are visible in the standard header files. The toggle *VP_UP_warn* is initialized to Off unless the warning level is set to 4.

Note: This mode is misnamed; strict ANSI conformance is enforced by the **-Hansi** option.

The default (**-Xt**) is determined by the driver configuration file, *hc.cnf*. The predefined macro `__STDC__` is set to 0 in **-Xa** and **-Xt** mode. Otherwise, it is set to 1.

Note: The use of long double is flagged as an error under options **-Xt** and **-Xa**.

- **-Xt:** The transition mode is used to compile pre-ANSI C programs. The compiler generally supports all new ANSI features and all High C extensions that are not incompatible with pre-ANSI C. It also relaxes enough of the ANSI rules to allow most pre-ANSI programs to compile unmodified.

The compiler warns about any construct that is illegal under ANSI but is permitted under pre-ANSI. The toggle *VP_UP_warn* is also turned On. See Chapter 4, “Compiler Toggles.”

If a language construct is encountered that has differing behavior between ANSI and pre-ANSI C, the pre-ANSI C interpretation is used. The toggle *Use_UP_rules* is turned On by default. This is the default mode.

- **-YI,dir:** Specify a directory to be searched last for include files.
- **-YP,dir1[:dir2[:dir3...]]]:** Specify default directories to be used for locating libraries.
- **-YS,dir:** Specify a directory containing the linker startup files.

- **-Zp [124]:** Forces packing within structures to be packed with no alignment holes between structure members (**-Zp** or **-Zpl**), with 2-byte alignment (**-Zp2**), or 4-byte alignment (**-Zp4**). The default is 4-byte structure padding, which is required by iABI.

Note that the `#pragma pack` option can be used within the C source file to pack specific structure definitions. See the section “Packed and Unpacked Structures” in Appendix F for information on packing structures.

Take care to insure that library or kernel interface structures are not packed, as these interfaces assume iABI structure padding.

Profiles

A profile is a file that is treated as a prefix to the source file. Its purpose is to customize the compiler to particular needs.

The default profile name is */usr/ccs/inc/hc.pro*, but an alternate name can be specified with the **-Hpro** command-line option (see the section entitled “Command-Line Options”). The option **-Hnopro** suppresses the reading of the profile.

You can use a profile to initialize toggles, specify aliasing conventions, and set other pragma options. These new “defaults” can then be applied to several or all source modules in a global fashion. In other words, the profile can be used to alter compiler defaults.

Example

A profile containing the following:

```
pragma On (List);
typedef long int integer;
```

would cause (1) the apparent default to be that a listing is produced, and (2) the type `integer` to be synonymous with `long int`.

Implementation

The compiler reads the profile immediately prior to reading the source file. It opens the profile in the same manner that it opens an Include or other input file. For more information, see Chapter 3, “Compiler Pragmas.”

CAUTION: If the profile is not found, the compiler immediately proceeds to read the source file without notice, even if **-Hpro** was specified on the command line. This occurs because **-Hpro** only changes the name to look for, but does not “demand” that the file be found.

Defaults File

A flexible way to set the compiler's default options is available in the form of the defaults file. The defaults file contains compiler options that are set prior to the processing of the command line options. These options establish the default behavior of the compiler but are overridden by command-line options.

This provides the user with more control over the compile process, and permits selective tailoring of the compile options. For example, to set the default optimization level for an environment where quick compile times are more important than code quality, the default optimization level can be set to zero by adding `-O0` in the defaults file. Likewise, the compiler can default to the strictly conforming ANSI C compile mode if the `-Xc` option is set in the defaults file.

The compiler looks for the file `/usr/ccs/lib/default.cc` when it starts up; and, if it is present, a line of options is read in. These options conceptually are prepended to the command line options, and so establish the default behavior unless a command line option overrides them. The shell variable `DEFAULT_DIR` can be set and exported to direct the compiler to look for `default.cc` in a directory other than `/usr/ccs/lib`.

The defaults file sets options for all compile sessions. To specifically set unique defaults for a session, the shell variable `DEFAULT_CC` can be set equal to the default options or to the path name to a defaults file and then be exported. Users can add a line to their login profile to set and export the compile environment default options needed. If the first character of `DEFAULT_CC` is a slash, it is interpreted as a path name to the defaults file; otherwise, the tokens are considered to be compiler options.

The order of evaluation used by the compiler to determine the defaults is to first check for a `DEFAULT_CC` shell variable. If this fails, the `DEFAULT_DIR` shell variable is used. Finally, `/usr/ccs/lib/default.cc` is used.

Examples of the contents of a *default.cc* file are:

- `-O0`
for quick compiles with little optimization.

- -586
for objects targeted with optimization for the Pentium processor.
- -W0 -O6 -686 -Hm=p6_only
to eliminate warnings, optimize at level 6, and generate Pentium Pro/Pentium II specific code by default.

Format Strings for Output-File Names

You can substitute *format strings* for parts of the explicit file names specified as arguments to the following command-line options that generate output files:

- Hbatch**
- Hefile**
- Hlist**
- Hmakeof**
- Hmap**

The format strings, listed in the table below, cause certain characteristics of the input file to be the default for the generated output file.

Table 2-1. Format Strings for Output File Names

Format String	Output-File Characteristic
@%d	Same directory pathname and file name as the input file
@%e	Same extension as the input file
@%f	Same full name as the input file: directory pathname, file name, and extension
@%n	Same file name as the input file
@%p	Same prefix as the input file: directory pathname, but not file name or extension

Example 1.

The following command line generates *myfile.lst*, a list file with the same file name as the input file (%n), with the extension *.lst*:

```
hc -Hlist=@%n.lst myfile.cpp
```

Example 2.

The command line below generates *myfile.xyz*, a batch file with the same file name as the input file (%d), with the extension *.xyz*, and puts it in the same directory as the input file:

```
hc -Hbatch=@%d.syz myfile.cpp
```


Compiler Pragmas

This chapter covers the following topics:

- Using Pragmas
- Pragma Reference
- Customizing Your Program with Pragmas

Using Pragmas

Compiler pragmas control toggle settings, linkage to external functions, data storage, inclusion and listing of source text, and other features.

Each pragma has a default value. You can supply a profile to *override* the defaults and thereby customize the compiler to a local environment (see Chapter 2, “Invoking the Compiler”).

Compiler pragmas take the following general form in your source:

```
pragma Pragma_name; /* or */  
pragma Pragma_name(Pragma_parameters);
```

where *Pragma_parameters* is a list of constant expressions separated by commas. The number and types of the expressions depend on the particular pragma. A pragma can appear anywhere a statement or declaration can appear. Pragma names are *not* case sensitive.

You can also specify pragmas on the command line with option **-Hpragma**:

```
-Hpragma=Pragma_name  
-Hpragma=Pragma_name(Pragma_parameters)
```

For more information about setting pragmas on the command line, see the option **-Hpragma** in the “Command-Line Options” section of Chapter 2, “Invoking the Compiler”.

Pragma Reference

Each pragma is described in the subsections below, providing the pragma syntax, parameter explanation, and action.

Pragmas

Alias(*Internal_name*, "*External_name*")

Assign an external name to an internal identifier.

Associates an external name with an internal name, for external or public purposes.

<i>Internal_name</i>	A variable or function identifier.
<i>External_name</i>	A string constant denoting the alternate or external name; must be enclosed in double quotes.

External_name does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker).

See the section “Aliasing External Names” later in this chapter.

Align_members([*n*])

Set the maximum boundary for structure-member alignment.

Specifies the maximum byte boundary on which an (unpacked) structure member must be aligned. *n* is an integer constant denoting the maximum boundary. (The default alignment is 4.)

Used without any arguments, pragma **Align_members** restores the default.

See the section “Aligning Data Structures” later in this chapter.

Called_infrequently (string [, string])

Designate “called infrequently” all functions whose names contain a specified string.

If a function is declared within the scope of this pragma, and if the function's name contains *string* as a substring, the compiler applies the calling-convention attribute `CALLED_INFREQUENTLY` to that function.

Whenever a function designated “called infrequently” is conditionally called within a loop construct, the compiler assumes that the basic block (node) containing the call is seldom executed. Based on that assumption, the compiler does the following:

- reorders the nodes so the node containing the function call is moved physically out of the loop
- sets a jump from the node containing the function call back into the loop

Pragma **`Called_infrequently`** must be specified *before* the declaration of the function(s).

Note: By default, the following names are used: `abort`, `error`, `fail`, `fatal`, or `trace`.

Optimization Benefits

Pragma **`Called_infrequently`** usually benefits error checking within a loop construct. When the portion of the `if` that contains the infrequently called function is moved out of the loop, the remaining code usually runs better, because there are less jumps involved. Such jumps would normally bypass the portion of the `if` that contains the infrequently called function.

This optimization usually benefits code (like a benchmark) that has hot spots. Hot spots are portions of code, usually loops, that run most of the time the code is running. This optimization also benefits database code.

Calling_convention ([Attributes [, _DEFAULT]])

Allow High C/C++ modules to call functions not compiled with High C/C++.

Controls linkage to subroutines by allowing calls to modules compiled with other C or C++ compilers, or in other languages such as assembly, Pascal, PL/M, or FORTRAN, from modules compiled with High C/C++.

- *Attributes* is a bit pattern in which each bit specifies a particular calling convention attribute.

- `_DEFAULT` sets the default calling convention.

Pragma **Calling_convention()** cancels the effect of **Calling_convention(Attributes)**.

See Chapter 8, “Externals”, for more information on calling external functions.

Ensure_instantiation(T1, T2,...Tn)

C++ only: Ensure instantiation of specified template types.

Ensures instantiation of the specified template types, regardless of the setting of toggle *Auto_class_member_instantiation*. Arguments *T1*, *T2*, ... *Tn* are template type identifiers.

For more information, see Chapter 11, “Compiling and Linking C++”.

Epilog_call("Proc_name")

Issue a call to *Proc_name* before a function return.

Causes the compiler to issue a call to the user-defined routine *Proc_name* just before a function’s return. *Proc_name* is the identifier of the function to be called; it must be enclosed in double quotes.

Exclude_instantiation(T1, T2,...Tn)

C++ only: Prevent instantiation of specified template types.

Prevents instantiation of the specified template types, regardless of the setting of toggle *Auto_class_member_instantiation*. Arguments *T1*, *T2*, ... *Tn* are template type identifiers.

For more information, see the section “Instantiating Templates Manually” in Chapter 11, “Compiling and Linking C++”.

Exit_expr(Expression [,Level])

Evaluate the *Expression* argument at program termination.

Evaluates the *Expression* argument when a program terminates. *Level* specifies the order in which expressions are evaluated. *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (zero), the default.

For more information, refer to the section later in this chapter entitled “Setting Module Initialization Priority”.

Global_aliasing_convention(["Form"])

Construct external object names for aliasing internal variable and function identifiers.

Specifies automatic construction of external object names for internal identifiers. *Form* is a sequence of substring substitutions and designators; it must be enclosed in double quotes. Pragma **Global_aliasing_convention()** cancels **Global_aliasing_convention(["Form"])**.

For more information, refer to the section later in this chapter entitled “Aliasing External Names”.

Ident("Comment")

Insert a comment in the generated object file.

Specifies a string to be inserted into the generated object module in the `.comment` section. If you specify option `-S`, the string is added with the `.ident` assembler directive. The comment string must be enclosed in double quotes.

For more information, see the section later in this chapter entitled “Placing Comments in Object Code”.

Initialization_level(Level)

Set initialization priority of modules.

Allows you to set the initialization priority for a module to *Level*, a constant integral expression whose value can be between 0 (zero) and 255, inclusive. The highest priority is 255; the default value of *Level* is 0 (zero).

For more information, see the section later in this chapter entitled “Setting Module Initialization Priority”.

Loop_factor(Loop_count, Size)

Unroll loops.

Optimizes code by unrolling loops; that is, it replaces a loop with the code that makes up the loop body, replicated some number of loop iterations.

- *Loop_count* is an integer constant specifying the maximum number of times a loop is unrolled; its default depends on the number of instructions in the loop.
- *Size* is an integer constant indicating the maximum size of loops to be unrolled.

If a loop contains a function call or switch statement, it will not be unrolled.

For more information, see the section “Loop Unrolling” in Appendix C, “Optimizing Program Performance”.

Off(Toggle_name)

Turn Off a toggle.

Turns Off a compiler toggle. See the section “What Is a Toggle?” in Chapter 4, “Compiler Toggles”.

Off_inline(function_name[, function_name ...])

Turn off inlining for specific functions.

When this pragma is active, it prevents calls to *function_name* from being inlined when they otherwise would be. This prevention occurs even if the function is declared as an inline function or (in the case of C++ class member functions) if the function is inlined in the class definition. The *Off_inline* state is pushed onto a stack for each function specified.

For more information, see pragmas **On_inline** and **Pop_inline**.

Offwarn(n1, n2, ...)

Disable specified warning messages.

Disables specified compiler warning messages (*n1*, *n2*, ...), so they are not displayed. (*n1*, *n2*, ...) are the numbers of the warning messages.

On(Toggle_name)

Turn On a toggle.

Turns On a compiler toggle. See the section “What Is a Toggle?” in Chapter 4, “Compiler Toggles”.

On_inline(function_name[, function_name ...])

Turn on inlining for specific functions.

When this pragma is active, it causes calls to *function_name* to be inlined even if a function does not otherwise meet the criteria for inlining (for example, if it is not declared as an inline function, or if it does not fit the size or call-count requirements for inlining of C functions).

Note: Even if a function appears in the **On_inline** argument list, it cannot be inlined if it contains calls to `Alloc()`, `alloca()`, or `setjmp()`, or if it is a *varargs* function. Pragma **On_inline** does, however, override inliner options that suppress inlining for recursive, exported, and nested functions, and functions that have their address taken.

The *On_inline* state is pushed onto a stack for each function specified.

For more information, see:

- pragmas **Off_inline** and **Pop_inline**
- Appendix D, “Inliner Function”

C++ only:

Pragma **On_inline** has no effect on inlinable C++ member functions. If C++ functions are to be inlined at all, they must be declared as inline functions or inlined directly in the class definition. Such functions are then inlined automatically, unless their inline status has been turned Off with pragma **Off_inline**.

With C-like functions in C++ code, pragma **On_inline** operates just as it does on regular C functions.

Onwarn(n1, n2, ...)

Enable specified warning messages.

Enables specified compiler warning messages (*n1*, *n2*, ...), so they are displayed. (*n1*, *n2*, ...) are the numbers of the warning messages.

Onwarn_error(*n1*, *n2*, ...)

Enable specified warning messages as compiler error messages.

Enables specified compiler warning messages, so they are displayed, and upgrades them to be compiler errors. (*n1*, *n2*, ...) are the numbers of the warning messages.

Pack([*n*])

Control default alignment of data structures.

See pragma **Align_members** and the section entitled “Aligning Data Structures” later in this chapter.

Page(*n*)

Insert page ejects in a source-code listing.

Causes page ejects to be inserted into a source-code listing. *n* is the number of form feeds (pages to eject). This pragma takes effect only when the *List* toggle is On. See Chapter 10, “Listings” for more information.

Pop(Toggle_name)

Reinstate prior status of a toggle.

Reinstates a prior status of a compiler toggle. See Chapter 4, “Compiler Toggles”.

Pop_align_members

Restore alignment prior to earlier **Push_align_members**.

Cancels previous **Push_align_members** and restores prior alignment. See the section later in this chapter entitled “Restoring Structure Alignment”.

Pop_ignore(Pragma_name, Pragma_name ...)

Reinstate prior ignore status of listed pragmas.

Restores the compiler's previous state of awareness of any pragma listed as one of its arguments. (Once a pragma has been set, you can make the compiler ignore it or become "unaware" of it by listing it as an argument of pragma **Push_ignore**.)

The compiler issues a warning if you "un-ignore" a pragma it is already aware of. The names of pragma arguments can be listed with or without quotes.

Pop_inline(function_name[, function_name ...])

Pop top-most stack entry for each specified function.

Pops the top-most stack entry for each function specified, and restores the previous **On_inline** or **Off_inline** state.

For more information, see:

- pragmas **On_inline** and **Off_inline**
- Appendix D, "Inliner Function"

Popwarn

Cancel previous pragma **Offwarn** or **Onwarn**.

Cancels the previous pragma **Offwarn** or **Onwarn**; these pragmas are "stacked".

Prolog_call("Proc_name")

Issue a call to *Proc_name* just after a function prolog.

Causes the compiler to issue a call to the user-defined routine *Proc_name* just after the function's prolog. *Proc_name* is the identifier of the function to be called; it must be enclosed in double quotes.

For more information, see:

- pragma **Epilog_call**
- toggles *Call_trace*, *Epilog_trace* and *Prolog_trace*

Push_align_members(n)

Set maximum boundary for structure-member alignment and save state.

Specifies that n bytes is the maximum boundary for structure member alignment, then saves the state for a subsequent call to pragma **Pop_align_members**.

To save the current state for pop, then restore the default, specify pragma **Push_align_members(_DEFAULT_MEMBER_ALIGNMENT)**.

For more information, see:

- section “Aligning Structures and Saving Maximum Alignment” later in this chapter
- pragma **Pop_align_members**

Push_ignore(Pragma_name, Pragma_name ...)

Ignore pragmas listed.

Causes the compiler to ignore (lose awareness of) any previously set pragma listed as one of its arguments. It also suppresses any warning message the compiler would otherwise emit about the pragma. **Push_ignore** is useful mainly for ignoring pragmas not supported by High C/C++. The names of pragmas to ignore can be listed with or without quotes.

Skip(n)

Insert blank lines in a source-code listing.

Causes blank lines to be inserted into a source-code listing. n is the number of blank lines to insert. This pragma takes effect only when toggle *List* is **On**. See the section entitled “Pragmas Page, Skip, Title” in Chapter 10, “Listings”.

Startup_expr(Expression[, Level])

Evaluate the *Expression* argument at start-up.

Evaluates the *Expression* argument at run-time start-up, before `main()` is invoked. If you use pragma **Startup_expr**, the compiler generates an initialization routine when it generates the object file.

Level specifies the order in which expressions are evaluated. *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (zero), the default.

See the section later in this chapter entitled “Evaluating Expressions Out of Scope of `main()`”.

Title("Listing_title")

Put a title at the top of each page of the source listing.

Causes a title to appear at the top of each page of the source listing. This pragma takes effect only when toggle *List* is **On**. *Listing_title*, a string constant, must be enclosed in double quotes. See the section entitled “Pragmas Page, Skip, Title” in Chapter 10, “Listings”.

Warning_level(n)

Set the level of warning diagnostics.

Sets the level of warning diagnostics to *n*, where *n* is a decimal integer between 0 (zero) and 4, inclusive..

Weak{x = y|(x,y)|x}

Unix targets only: Designate a weak global symbol and assign it a value.

Designates identifier *x* as a weak global symbol whose value is the same as identifier *y*. Identifier *y* must be a global symbol defined in the current module with *x* otherwise undefined. Pragma **Weak x** is equivalent to both **Weak(x,x)** and **Weak x = x**.

Customizing Your Program with Pragmas

Aligning Data

Aligning Structure Members

Use pragma **Align_members** to specify how members of subsequently declared structures are to be aligned. This pragma has the form:

```
#pragma Align_members(n)
```

where *n* is a decimal integer constant denoting in bytes the *maximum* alignment boundary to be applied to a member. The default setting for **Align_members** is 4.

Here is an example use of pragma **Align_members**:

```
#pragma Align_members(2)
struct s {
    char c;      /* Offset 0, one-byte padding */
    int x;       /* Offset 2 */
};
```

Member *x*, which is type `int`, would ordinarily be aligned to a four-byte boundary. But because of the pragma **Align_members**, *x* is aligned to a two-byte boundary instead.

Note that pragma **Align_members** denotes the maximum boundary; an `int` member is never aligned to greater than a four-byte boundary regardless of how large a value is specified for *n*.

Packing a *struct*

To pack a `struct` so that no padding is inserted between members, specify a setting of 1:

```
#pragma Align_members(1)
```

WARNING: If you use pragma **Align_members** indiscriminately, you can severely impair run-time performance by trying to access `struct` members that are not properly aligned.

Aligning Data Structures

Pragma *Pack* is synonymous with pragma **Align_members** — it controls the default alignment of data structures.

- Pragma **Pack(1)** packs a struct so that no padding is inserted between members.
- Pragma **Pack()** “unpacks” the data; that is, it restores the alignment to the default (“natural”) alignment of data structures on the target machine, as specified by the application binary interface.

Note: Pragmas **Pack(n)** and **Pack()** are not stackable. **Pack()** does not revert to the previous **Pack(n)**, so these pragmas cannot be nested. For nesting, use pragma **Push_align_members** and **Pop_align_members** pairs.

See the section below entitled “Aligning Structures and Saving Maximum Alignment” for an example of when *not* to use pragma **Pack(1)/Pack()** pairs.

Aligning Structures and Saving Maximum Alignment

Pragma **Push_align_members(n)** has the same effect as pragma **Align_members(n)** or pragma **Pack(n)**, except that **Push_align_members** saves the current setting of the maximum `struct` alignment.

Figure 3-1 illustrates why we recommend the use of pragma **Push_align_members(1)** and **Pop_align_members** pairs (because they are “stackable”) rather than pragma **Pack(1)** and **Pack()** pairs (which are not stackable).

In the following figure, structure `first_st` is naturally aligned, while structure `second_st` is packed (aligned to 1).

Figure 3-1. Using Push_align_members/Pop_align_members Pairs

```

/* Pack structure */
#pragma Push_align_members(1)      /* push #1 */
/* --- Begin included header file --- */
/* ... */
/* Align to machine natural boundary */
#pragma Push_align_members(4)      /* push #2 */
struct first_st {
    char a, int b, char c;
}
/* Revert to previous alignment */
#pragma Pop_align_members()        /* pop #2 */
/* --- End included header file --- */
struct second_st {
    char a, int b, char c
}
/* Revert to previous alignment */
#pragma Pop_align_members()        /* pop #1 */

```

If you replace the “push #1/pop #1” and “push #2/pop #2” pragma pairs in Figure 3-1 with **Pack(n)/Pack()** pairs, as shown in Figure 3-2, structures `first_st` and `second_st` will both be naturally aligned.

Figure 3-2. Pack(n)/Pack() Pairs

```

/* Pack structure */
#pragma Pack(1)                    /* push #1 */
/* --- Begin included header file --- */
/* ... */
#pragma Pack(4)                    /* push #2 */
struct first_st {
    char a, int b, char c;
}
/* Does not revert to previous alignment */
#pragma Pack()                     /* pop #2 */
/* --- End included header file --- */
struct second_st {
    char a, int b, char c;
}
/* Does not revert to previous alignment */
#pragma Pack()                     /* pop #1 */

```

This natural alignment occurs because the alignment setting is changed in the included header file, and **Pack()** does not revert to the previous alignment setting (but **Pack()** always restores the alignment setting to the natural alignment setting of the target machine).

Restoring Structure Alignment

Pragma **Pop_align_members** cancels **Push_align_members** and restores the alignment that existed before the preceding **Push_align_members**.

If there is no pending **Push_align_members**, the alignment is restored to the default.

Using Initialization Pragmas

Setting Module Initialization Priority

Pragma **Initialization_level(Level)** sets the initialization level for the module to the specified value.

The program start-up code calls each of the initialization routines in order of decreasing initialization level, where *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive.

If you do not specify *Level*, the compiler assumes a value of 0 (zero is the lowest priority).

Note: Initialization priorities in the range 120 through 130 are reserved.

Evaluating Expressions Out of Scope of main()

You can specify expressions to be evaluated at program start-up or termination with pragmas **Startup_expr** and **Exit_expr**. If you specify **Startup_expr** or **Exit_expr**, the compiler generates an initialization or termination routine, respectively.

Use these initialization pragmas if you want some code to be executed before **main()** is called or after **main()** exits. Such code could perform initialization of data structures and could free memory to undo the initialization process.

If data items must be initialized at run time, or expressions must be evaluated at start-up or termination, the compiler generates an initialization routine when it generates an object module.

Pragma **Startup_expr**

Pragma **Startup_expr(Expression, Level)** evaluates the *Expression* argument at run time before `main( )` is invoked. The order in which expressions are evaluated is specified by the integer constant expression *Level*.

If you do not specify *Level*, the compiler uses either the initialization level of the entire compilation unit, or the default value 0 (zero). (Use pragma **Initialization_level** to set a compilation unit's initialization level, which is 0 (zero) by default.)

Typically, the *Expression* argument is a function call:

```
void initialize_this_module() { ... }
#pragma Startup_expr( initialize_this_module(), 10 )
```

With High C/C++, the full power of the expression is available, and can be used to disambiguate between two functions or to execute other non-function code at start-up:

```
void init(int), init(double); int x, y;
/*
 * C++ code here
 */
#pragma Startup_expr( (init(3), init(2.71),
                     x = 4, y = x*x), 10)
```

To specify more than one expression, enclose them in parentheses, or use multiple pragmas.

Pragma **Exit_expr**

Pragma **Exit_expr(Expression, Level)** evaluates the *Expression* argument when the program terminates, at the time `atexit( )` gains control.

The integer constant expression *Level* specifies the order in which expressions are evaluated. If you do not specify *Level*, the compiler uses the default value 0 (zero).

Aliasing External Names

An *aliasing convention* is a method for aliasing internal names to external names. The names of variables and functions that are communicated across module boundaries are normally made global in the resultant object module. In large programs there can be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately, C compilers and most linkers do not provide for a structured name space, such as named packages of resources. Thus the well-chosen internal names in a program might not be usable as external names (those known to the linker). Some method of aliasing internal names to external names is needed, and High C/C++ provides it.

It is important to be able to alias such names to avoid conflicts in the linker's external symbol dictionary, rather than being forced to *pervert* the internal names themselves. The internal names must be well-chosen “containers of meaning,” for program maintainability.

The High C/C++ compiler provides two pragmas for aliasing names: **Alias** and **Global_aliasing_convention**.

Pragma Alias

Pragma **Alias** specifies, for a given internal name, another name for external or public purposes, as follows:

```
#pragma Alias(Internal_name, "External_name")
```

External_name does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker).

The **Alias** pragma must appear *in the scope of* the declaration of the internal name. For example:

```
void Initialize();
#pragma Alias(Initialize, "x_initialize")
/*
 * The function Initialize() is referenced
 * in the object-module symbol table as
 * "x_initialize"
 */
int BA;
#pragma Alias(BA, "Ph.D.")
/*
```

```
* "BA" is referenced in the
* symbol table as "Ph.D."
*/
```

Pragma **Global_aliasing_convention**

Pragma **Global_aliasing_convention** specifies the aliasing of external names not already aliased by the **Alias** pragma, as follows:

```
#pragma Global_aliasing_convention("Form")
```

Form is a string that specifies the way each *External_name* is derived from its *Internal_name*.

External_name is the actual text of *Form*, except for the following substitutions for substrings within *Form*:

- %r (resource) denotes *Internal_name*
- %% denotes %
- %C, %c, and %a affect the casing of the external name

For example, if you specify

```
#pragma Global_aliasing_convention("%_r")
```

a function named `fred` will be known as `_fred` in the object module. See the section below, “Arguments to `Global_aliasing_convention`”, for additional information.

Setting the convention in code

If you set the aliasing convention *in the source code or in a profile*, only subsequently declared names are affected. Functions that are used but not declared are *not* affected by the convention; their external names have the “_” prefix.

Overriding an aliasing convention

You can override the **Global_aliasing_convention** setting for a particular name by specifying pragma **Alias**; that is, pragma **Alias** takes precedence.

Turning Off the pragma

You can turn **Off** pragma **Global_aliasing_convention** by omitting the argument:

```
#pragma Global_aliasing_convention()
```

This restores the original default.

Arguments to `Global_aliasing_convention`

The `%r` argument can be followed by substring designators of the form `:Start:Len` to indicate the substring starting at character *Start* and going for *Len* characters. The character positions are numbered starting at 1 (one).

- If *Start* is negative, it denotes the starting position from the *right end* of the string where the last character is numbered -1.
- If you omit *:Len*, the substring means “to the end of the string.”

These substitutions within Form affect the casing of the external name:

- `%C` means to shift subsequent letters in the external name to uppercase.
- `%c` means to shift them to lowercase.
- `%a` means to use subsequent letters as is, each with its given case (that is, do not shift case). This mode holds at the beginning of the specification.

Examples:

- Change all global names (`%r`) to uppercase (`%C`) and truncate them to eight characters (`:1:8`):

```
#pragma Global_aliasing_convention("%C%r:1:8")
```
- This same example would appear on the command line as:

```
-Hgac="%C%r:1:8"
```

Placing Comments in Object Code

Use pragma **Ident** to place notes in a comment section within a generated object module, as follows:

```
#pragma Ident("comment string")
```

You can use this pragma multiple times. The name of the comment section is typically `.comment`. When the compiler generates an assembly file, the pragma typically translates to an `.ident` statement.

The compiler normally emits a comment string that identifies the version of the compiler and the selected command-line options used to produce the object file, even when you do not use the **Ident** pragma.

This string can be suppressed by turning Off the *Command_line_ident* toggle.

These are some example uses of the **Ident** pragma:

```
#pragma Ident("Source version 3.4.1\n")  
#pragma Ident("Date is " _ _DATE_ _)
```


Compiler Toggles

This chapter includes the following topics:

- What Is A Toggle?
- Values and Descriptions of Compiler Toggles

What Is A Toggle?

A toggle enables an optional compiler feature. For example, `List` enables or disables program listings.

A toggle can be set on the command line with option **-Hon** or **-Hoff**; the syntax is:

```
cc -Hon=Toggle_name
```

or

```
cc -Hoff=Toggle_name
```

Toggles can also be turned On or Off with pragmas placed in source code or in a profile (see Chapter 3, “Compiler Pragmas”). The syntax is:

```
#pragma On(Toggle_name);  
#pragma Off(Toggle_name);  
#pragma Pop(Toggle_name);
```

`On` turns the toggle on; `Off` turns it off; and **Pop** reinstates it to a prior value. Toggles operate in a stack-like fashion, where each `On` or `Off` is a “push” of `On` or `Off`, and a **Pop** “pops” the stack. The stack for each toggle is at least 16 elements deep, but no diagnostic is given if the stack overflows or underflows.

For example, the following toggles turn the source listing Off and On.

```
#pragma On (List); – Turns on the source listing.  
#pragma Off(List); – Turns off the source listing.  
#pragma On (List); – Turns on the source listing.  
#pragma Pop(List); – Back to off for the listing.  
#pragma Pop(List); – Back to on for the listing.
```

Recall that toggles can also be initialized on the command line, with **-Hon** and **-Hoff**. See Chapter 2, “Invoking the Compiler.”

Note: Toggle names are not case insensitive.

Values and Descriptions of Compiler Toggles

The names, default values, and meanings of the compiler toggles are described in this section.

Compiler Toggles

387

Default: On for SCDE 3.02 and Off for earlier releases

When this setting is On, the compiler generates in-line 80387 code for the C library functions `sin`, `cos`, `tan`, `atan2`, `atan`, `acos`, and `asin`. In addition, `sin` and `cos` of the same argument, when they are adjacent, are computed by a single 80387 instruction.

CAUTION: Be sure `387` is turned On before any declarations in the program are processed. Turning `387` on generates faster code, but limits the degree of error handling done. For full UNIX error handling, leave this option Off.

ABI_Args_in_regs

Default: On with -O level 3 or higher

Generates two entry points for exported functions, one a conventional ABI compliant entry point and a arguments in registers entry point for internal calls within a source module.

Aggressive_cloning

Default: Off

When the **-Hbbarc** option is used, the SCDE-3.01 and later compilers invoke a new basic block ordering algorithm, which clones certain small basic blocks which are frequently executed so as to minimize the number of "taken" branches. The *aggressive_cloning* cloning toggle reduces the cloning thresholds as a result cloning more basic blocks and further reducing the number of "taken" branches. It should be noted that when this toggle is switched On, the size of the application's binary may considerably increase, thereby polluting the cache. Thus, it is recommended that this toggle be switched On only for small applications.

Align_all_labels

Default: Off

When this setting is On, all labels are aligned on four-byte boundaries by inserting NOPs appropriately. This can improve performance in applications that spend much of their time in tight loops.

Align_labels

Default: On

When this setting is On, labels that are jumped to but never "fallen through" to, are aligned on four-byte boundaries. Examples are labels that are preceded by an unconditional jump, such as the label for the else part of an if-then-else statement. This typically results in a speed improvement by reducing the time for a jump instruction, at the expense of padding bytes inserted to effect the alignment.

Note: This toggle is turned Off when toggle *Optimize_for_space* is On.

Align_routines

Default: On

When this setting is On, function entry points are aligned on four-byte boundaries.

Note: This toggle is turned Off when toggle *Optimize_for_space* is On.

Arithmetic_simplification

Default:

- Off if optimization level = 0 - 2, or if you specify option **-g**
- On if optimization level = 3 - 7

Perform arithmetic simplification optimization across entire function.

When toggle *Arithmetic_simplification* is On, the compiler performs arithmetic simplification across the entire function currently being compiled. Many of these simplifications occur during local common sub-expression elimination, but are not performed on a global level.

For example, in the following code sequence:

```
int doit(int b,int c) {
    int a = b*c;
    do_something_else;
    return a/c;
}
```

if toggle *Arithmetic_simplification* is On, the compiler optimizes away the multiply and divide instructions and returns the value *b*.

If toggle *optimize_floating_point* is On, toggle *Arithmetic_simplification* causes the compiler to also perform floating-point simplifications.

Normally, the ANSI C Standard requires that floating-point operations be performed exactly as specified, as the results of the operation might be contingent upon the order in which the expressions are evaluated. When toggles *Arithmetic_simplification* and *optimize_floating point* are On, the compiler relaxes the rules associated with normal floating-point operations.

Toggle *Arithmetic_simplification* has no effect if you specify option **-g** or if you turn Off toggle SSA.

Args_in_regs_for_locals

Default: On

When this setting is On, the `ARGS_IN_REGS` calling-convention attribute is used for a function *F* provided the following conditions are met:

- *F* is local (that is, “static”).

- F's address has not been accessed (that is, F can be invoked only by a direct call).

The 486 toggle is On (ARGS_IN_REGS has meaning only in 486 mode).

For a description of the ARGS_IN_REGS attribute, see the “Calling-Convention Attribute Specified with Function Declaration” section in Appendix F, “High C Language Extensions”.

Asm

Default: Off

Turn this toggle On if you want an assembly listing. When this setting is On, a (pseudo-)assembly listing is generated, annotated with source code as assembly comments. If the *Asm* toggle is to be turned On and Off over sections of a module, On-Off pairs should surround function definitions; if the pragma is used among executable statements, the point at which the pragma takes place may be obscured by the use of optimizations.

Auto_class_member_instantiation

Default: On

C++ only: Enable automatic instantiation of class templates.

In a C++ program, a class template must always be instantiated when it is used. By default, the compiler generates instances of class members when the class is used in a source module, and when the definitions of those members are likewise presented to the compiler. For more information, see the section “The Single-Copy Problem” in Chapter 11, “Compiling and Linking C++”.

To disable automatic instantiation, turn off toggle *Auto_class_member_instantiation*.

Note: Toggle *Auto_class_member_instantiation* has no effect if you use the COMDAT solution.

See the following for more information:

- Chapter 11, “Compiling and Linking C++”, sections:
 - “The Single-Copy Problem”

- “Using COMDAT Sections to Merge Global Instantiations”
- “Instantiating Templates Manually”
- option **-Hnocomdat** in Chapter 2, “Invoking the Compiler”
- toggle *Auto_func_instantiation* below

Auto_func_instantiation

Default: On

C++ only: Enable automatic instantiation of function templates.

When you use the same template in separate source modules in a C++ program, the linker returns an error complaining of a duplicate definition.

To prevent duplicate definitions, disable automatic template instantiation by turning off toggle *Auto_func_instantiation*.

Note: Toggle *Auto_func_instantiation* has no effect if you use the COMDAT solution.

See the following for more information:

- Chapter 11, “Compiling and Linking C++”, sections:
 - “The Single-Copy Problem”
 - “Using COMDAT Sections to Merge Global Instantiations”
 - “Instantiating Templates Manually”
- option **-Hnocomdat** in Chapter 2, “Invoking the Compiler”
- toggle *Auto_class_member_instantiation*

AutoDebug

Default: Off

C++ only: Place named SDI for a named struct in the struct's debug repository.

When toggle *AutoDebug* is On and you specify option **-g**, the compiler places named symbolic debug information (SDI) in the debug repository of any named C or C++ *struct* (including unions and classes) declared within the toggle's scope.

The debug repository of a *struct* is the compilation unit in which the *struct*'s first non-pure-virtual, non-inline function is defined.

Back_substitute_epilog

Default: Off unless optimization level ≥ 2 or `-Oc` is set.

When this setting is On, the compiler replaces an unconditional jump to the epilogue of a function with the epilogue sequence.

Behaved

Default: Off unless optimization level ≥ 4

When the setting is On, this toggle specifies that the program is “well behaved,” and the optimizer can be less conservative in generating code for pointer-based objects. A well behaved program is one that follows these rules:

- The address of a union member is never assigned to a pointer.
- A value of a pointer type is never case to an incompatible pointer type.

With these assumptions, the compiler might be able to generate substantially better code in referencing pointer-based variables.

The compiler issues an appropriate warning if any of the above assumptions are violated in such a way as to affect assumptions made by the optimizer. You must decide whether the warnings can be safely ignored or whether the program should be compiled at a lower optimization level.

CAUTION: The compiler might not catch all instances of misbehaved code. For example, a pointer-to-char may be passed to an undeclared external function expecting a pointer-to-int. Therefore, it is possible for a program to compile at optimization level 4 without warnings (and run incorrectly), but run correctly when compiled at a lower optimization level.

Borland

Default: Off

C++ only: Relax some ARM C++ standards to conform to the Borland class library.

In C++ compilations, toggle *Borland* relaxes some compile-time error checking, and ensures that the type of *sizeof* is the same as *size_t*.

The following operations do not conform to the *Annotated C++ Reference Manual*, but they are relaxed when this toggle is On because they appear in Borland class-library source:

- ANSI Standard C requires the type of *sizeof* to be the same as *size_t*. However, *size_t* is *unsigned*, and code such as this:

$$\text{if (amount-left - sizeof(item-to-allocate)) > 0}$$
does not work if the type of *sizeof* is *size_t*. In particular, for High C/C++, *sizeof* is *signed*.
- In a conditional expression such as $x? Y: z$ or $x? z : Y$, the *Y* can be a *const structure type T*, and *z* a non-*const T*, the result type is *T*.

Callee_pops_when_possible

Default: On

When this setting is On, the compiler uses the CALLEE_POPS_STACK calling convention attribute for function definitions that meet the following criteria:

- The function is static.
- The address of the function is not assigned to a pointer nor passed as a parameter.
- The function has not been declared with variable arguments.
- The function is prototyped or called consistently with the same number and types of parameters.

Note: Leave this toggle On to save code space and improve performance.

Char_default_unsigned

Default: Off

When this setting is On, type *char* is unsigned by default.

The ANSI Standard allows the type `char` by itself; that is, without the adjectives `unsigned` or `signed`, to be either signed or unsigned. Of course, the types `unsigned char` and `signed char` can be used to explicitly control signedness.

CAUTION: The i386 ABI defines `char` to be signed. To be compliant, do not use *Char_default_unsigned*.

Char_is_rep

Default: Off

The ANSI Standard provides three character data types: `char`, `unsigned char`, and `signed char`. Even though `char` is signed or unsigned, depending on the implementation, it is never the same type as `signed char` or `unsigned char`. One of the effects is that `char*` is type compatible with neither `unsigned char*` nor `signed char*`. This increases program portability.

When this setting is On, *Char_is_rep* specifies that `char` is identically either `signed char` or `unsigned char`, depending on whether `char` is signed. Although this may permit a program to compile, the program might not compile under another High C implementation where the signedness of `char` is different. Be sure to turn this toggle On before any declarations or preprocessor statements.

Cleanup_spills

Default: Off unless optimization level ≥ 3 or `-Oc`

When this setting is On, the compiler performs another iteration of local common subexpression elimination to clean up any register spill code if possible. See the “Level 0 Optimizations - Local Common Subexpression Elimination” section of Appendix C, “Optimizing Program Performance” for more information.

Command_line_idnt

Default: On

When this setting is On, the compiler inserts information into a comment section in the generated code. This information includes the version of the compiler and command-line options that were specified (SCDE Release 2.01 or later).

Cross_jump

Default: On

When Off, suppresses the cross-jumping (tail-merging) optimization that is otherwise performed when **-O** is specified. Suppressing cross-jumping can make code easier to debug at the instruction level.

Define_static_members

Default: On

C++ only: Supply implicit definition of static class data members.

According to the C++ language definition, static data members of a class must be defined outside the class.

```
class s {
static int x; /* This is NOT a definition */
};
int s::x;      /* This is the required definition */
```

Turning On toggle *Define_static_members* relaxes this restriction, and the definition is implicitly supplied by the compiler. The toggle accommodates some existing practices.

See Chapter 1, “Features of High Performance C/C++”.

Demote_sizes

Default:

- Off if optimization level = 0 - 4, or s, or if you specify option **-g**
- On if optimization level = 5 - 7

Allow **==** operator to compare integer types other than **int**.

When toggle *Demote_sizes* is On, operator **==** can take operands of type *unsigned short*, *signed short*, and *char* without casting them to *int*.

Double_return

Default: Off

When this setting is On, any non-prototype functions returning type float returns type double instead. This convention conforms to some Portable C Compiler implementations. This is the default behavior for **-Xt** compile mode.

Emit_names

Default: Off

Precede function definition code with function name and name size.

When `-toggle Emit_names` is On, the compiler precedes the generated code for each function definition with the name of the function.

The length of the name is put in the byte immediately preceding the function's entry point.

```

    byte    0x6d, 0x61, 0x69, 0x6e
    byte    0x4
fmc:...
```

This information can help with interpreting memory dumps.

Empty_is_void

Default: On

C++ only: Force C++ (void) interpretation of function declarations with no arguments.

In C++, a function declared `int f();` means the same as `int f(void);`; the function takes no arguments. In C this is not the case; a function declared `int f();` can take any arbitrary arguments.

When Off, toggle `Empty_is_void` forces the C, not C++, interpretation of such function declarations. This is useful in "wrapping" old C header files that might contain such declarations, when compiling with C++:

```

extern "C" {
#pragma Off(Empty-is-void)
#include "old-hdr.h"
#pragma Pop(Empty-is-void)
```

```
}
```

Toggle *Empty_is_void* is turned Off in *c_wrap1.h*.

Enforce_access_control

Default: On

C++ *only*: Allow compiler-enforced access control.

In C++ compilations, turn Off toggle *Enforce_access_control* to remove all compiler-enforced access control. This removal can be useful when, for example, you are in a debugging mode and want to print out some private members of a class. Normally, access control might deny you access to those members.

Ensure_instantiation

Default: Off

C++ *only*: Force instantiation of all template types.

Toggle *Ensure_instantiation* forces instantiation of any template types used within its scope, whether or not toggle *Auto_class_member_instantiation* is turned On.

See also the related pragmas, **Ensure_instantiation** and

Exclude_instantiation, described in Chapter 3, “Compiler Pragmas” and in Chapter 11, “Compiling and Linking C++”.

Epilog_trace

Default: Off

Generate information about a function being exited.

When toggle *Epilog_trace* is On, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is exited.

For more information, see toggles *Call_trace* and *Prolog_trace*.

Exception_aware_class

Default: Off

C++ only: Compile exception-aware classes.

Toggle *Exception_aware_class* causes all classes compiled within its scope to be exception-aware. That is, whenever such a class is constructed, its constructor increments the global subobject count by 1. When such a class is destroyed, its destructor decrements the global subobject count by 1.

Exception-aware classes permit High C++ exception-handling (as distinct from system exception-handling).

See the section “Using Exception Handling” in Chapter 11, “Compiling and Linking C++” for details on exception handling.

Export_vtabs

Default: Off

C++ only: Make virtual function tables public and defined.

Toggle *Export_vtabs* affects C++ classes that contain only inlined and pure virtual functions. If *Export_vtabs* is On, the virtual-function tables for such classes are made public and defined, and can be referenced from any module compiled with toggle *Import_vtabs* turned On.

Use toggle *Export_vtabs* in conjunction with toggle *Import_vtabs* to force only one copy of virtual-function tables for these classes to be defined in a program.

See also toggle *Import_vtabs*.

Fast_virtual_bases

Default: Off

C++ only: Speed access to members of virtual bases.

Turning On toggle *Fast_virtual_bases* can improve speed of access to members of virtual bases of a class when a class hierarchy contains virtual bases as bases of other virtual bases.

When a virtual-base class **B** is accessible only as a base of another virtual-base class, access to **B** requires two pointer indirections. In general, multiple pointer indirections can be required for more complicated class hierarchies.

One way to improve the speed of access is to turn on toggle *Fast_virtual_bases*. With the toggle on, the compiler guarantees that a class has enough copies of virtual-base pointers directly accessible within the class so that access to any virtual base of the class requires only a single pointer indirection.

Toggle *Fast_virtual_bases* can be turned On and Off around class definitions; turn it on just before the definition of a class for which you want fast virtual-base access.

You can modify your C++ source to use fast virtual-base access on a case-by-case basis. For example:

```
class B { int x; };
class X : virtual B {      };
class D : virtual X {      };
D *dp;
```

Here, `dp->x` would require two virtual-base pointer indirections; the first to access `x` from `D`, and the second to access `B` (and thus `x`) from `X`. You can use the toggle, as just described:

```
class B { int x; };
class X : virtual B {      };
#pragma On(Fast_virtual_bases)
class D : virtual X {      };
#pragma Pop(Fast_virtual_bases)
D *dp;
```

But you can also modify the class definition of `D` to specify `B` as a direct virtual base of `D`. Direct virtual bases are always accessible via one virtual-base pointer indirection.

```
class B { int x; };
class X : virtual B {      };
class D : virtual X {      }, virtual B {      };
D *dp;
```

In each of the last two examples, `dp->x` requires only one virtual-base pointer indirection.

CAUTION: If you leave toggle *Fast_virtual_bases* turned on indiscriminately, your program requires more space.

Fwait_before_return

Default: Off

Generate an fwait instruction before returning int functions that do i387 operations. This will force an exception to occur in the function where the computation occurred, instead of further down the instruction execution stream.

Global_CSE

Default: Off unless optimization level ≥ 2 or -Oc

When this setting is On, the compiler performs global common subexpression elimination. See the “Level 2 Optimizations - Global Common Subexpression Elimination” section in Appendix C, “Optimizing Program Performance” for more information.

CAUTION: Do not turn On toggle *Preload_args_from_memory* when toggle *Global_CSE* is On.

Globals_volatile

Default: Off unless -Xt is specified

When this setting is On, the optimizer considers global and local statically-defined variables as volatile.

Note: Turn this toggle On if your program relies on signal functions to modify static or global variables. You can turn this toggle On and Off between function definitions.

High_level_scheduling

Default: Off

When this setting is On, the compiler performs instruction scheduling before it performs register allocation. See the “Level 3 Optimizations - Instruction Scheduling” section in Appendix C, “Optimizing Program Performance” for more information.

Import_vtabs

Default: Off

C++ only: Assume virtual function tables to be external for classes with inlined, pure virtual functions.

In C++ compilations, toggle *Import_vtabs* applies to classes that contain inlined, pure virtual functions. If *Import_vtabs* is On, the virtual-function tables for such classes are assumed to be defined in some other module. Use toggle *Export_vtabs* to force this definition in the defining module.

Use toggle *Import_vtabs* in conjunction with toggle *Export_vtabs* to force only one copy of virtual-function tables for these classes to be defined in a program.

For more information, see the toggle *Export_vtabs*.

Improved_spill_analysis

Default:

- Off optimization level = 0 - 5, or s, or if you specify **-g**;
- On if optimization level = 6 or 7

When toggle *Improved_spill_analysis* is On, the compiler optimizes the spill code generated by the register allocator.

Induction_analysis

Default: Off if optimization level = 0–4. or s; or if optimization level = 5–7

Readjust loop counter to eliminate compares.

When toggle *Induction_analysis* is On, the compiler analyzes loops, and in some cases readjusts the loop counter to eliminate the compare.

Init_stack

Default: Off

Help catch uninitialized pointers.

When toggle *Init_stack* is On, it causes the compiler to generate code automatically at each function prolog to initialize all auto variables to the hexadecimal value `0xdead_beef`. Because this value is usually outside the range of your program, this is a good way to catch uninitialized pointers.

Inline_common

Default: On

C++ *only*: Solve the single-copy problem for inline functions (COMDAT).

When toggle *Inline_common* is On, it solves the single-copy problem for the following types of inline functions:

- those whose bodies must be generated (for example, when the function is too big to inline, or when its address is taken)
- those whose bodies are defined by the user within the scope of the toggle

This includes inline functions within classes, and global functions declared `extern inline`.

Note: Toggle *Inline_common* is turned On when option **-Hcomdat** is specified (the default). The toggle is turned Off when option **-Hnocomdat** is specified.

For a discussion of the single-copy problem and its solutions, see the sections “Using Templates” and “Instantiating Templates Manually” in Chapter 11, “Compiling and Linking C++”.

Int_function_warnings

Default: Off

When Off, suppresses the warning message normally generated when a function returning `int` has no `return` `exprn`; statement within it, or when a function returning `int` contains a `return`; within it.

This is to remove frequent warnings for old C source that did not use the reserved word `void` to indicate a function returning no result, because such functions return `int` by default.

Interprocedural_optimizations

Default:

- Off if optimization level = 0 - 5, or s, or if you specify option **-g**;
- On if optimization level = 6 - 7

Determine best possible order for generating code for functions in a module.

When toggle *Interprocedure_optimizations* is On, the compiler determines the best possible order for generating code for the functions presented in a source module; that is, it optimizes across function boundaries.

See Appendix C, “Optimizing Program Performance”, for more information.

List

Default: Off

When this setting is On, the compiler produces a listing on standard output. Usually this toggle is set at the start of compilation, but it can also appear in the source file to turn the listing On or Off around a particular section of source.

Literals_in_code

Default: Off

Put literals in code space instead of in data space.

When toggle *Literals_in_code* is On, literals in a program are placed in the code space rather than in the data space. This placement is beneficial if the operating system performs dynamic code loading.

In such circumstances, code is more often read-only, so literals can be swapped out of memory without having to be written to the paging medium used by the operating system.

Live_dead_iterate

Default: Off unless optimization level ≥ 2

When this setting is On, the compiler performs multiple iterations of live/dead analysis to further improve the code. See “Level 0 Optimizations - Live/Dead Analysis” section in Appendix C, “Optimizing Program Performance”, for more information.

Local_CSE_iterate

Default: Off unless optimization level ≥ 2

When this setting is On, the compiler performs multiple iterations of local common subexpression elimination to further improve the code. See the “Level 0 Optimizations - Local Common Subexpression Elimination” section in Appendix C, “Optimizing Program Performance”, for more information.

Long_enums

Default: On

When this setting is On, any variable of an enum type is mapped to an int-sized region of memory. Otherwise, such a variable is mapped to the smallest of one, two, or four bytes, so that all values are representable.

Loop_reg_rename

Default: Off unless optimization level ≥ 2

When this setting is On, the compiler performs register lifetime analysis to improve register allocation. See “Level 2 Optimizations - Register Lifetime Analysis” in Appendix C, “Optimizing Program Performance”, for more information.

Low_level_scheduling

Default: Off unless optimization level ≥ 3

When this setting is On, the compiler performs instruction scheduling after it performs register allocation. See “Instruction Scheduling” in Appendix C, “Optimizing Program Performance”, for more information. This only applies to -486 and -586 (SCDE Release 2.01 or later).

Make_externs_global**Default:** On

Make objects with storage class `extern` be global.

When toggle *Make_externs_global* is On, any local declaration of an object with storage class *extern* is made global if there is not already a global declaration of the object. Early C compilers promoted an `extern` declaration within a function to the global scope. This toggle supports programs depending upon that feature.

Mem_refs_from_loop**Default:** Off unless optimization level ≥ 2

When this setting is On, the compiler attempts to replace variable memory references in loops with references to registers. See the section “Loop Memory Reference Elimination” in Appendix C, “Optimizing Program Performance”, for more information. (SCDE Release 2.01 or later).

NCR_predict_order**Default:** On

When this setting is On, the compiler performs Static Branch Prediction. The compiler analyzes branch conditions based on a number of heuristics. Branches which are unlikely to be taken are moved out of the main control flow of the program. An example of a branch which would not usually be taken would be a call to an error handling routine.

NCRsetcc**Default:** On for -486, -586, and -686

Use the iX86 setcc instruction instead of if expansion for logical I/O generation.

NCRsuperblock**Default:** Off

Used in conjunction with **-qb** and **-Hbbarc**, currently clones or duplicates program segments to eliminate branching. See the “Types of Basic Block Optimizations” section in Appendix C, “Optimizing Program Performance”.

NCRswitch

Default: On

This optimization uses the program statistics obtained by previously profiling the application to duplicate one or two of the most frequently executed “CASEs” of a switch statement.

This optimization can be used only in conjunction with the **-Hbbarc=** option.

Nested_types

Default: On

C++ *only*: Make nested types invisible outside the class where they are declared.

The C++ language includes nested types, meaning that types declared within the scope of a class remain within that scope only and are not visible outside the class.

```
class s { typedef int T; };
T x;      // Error: T is undeclared
```

In pre-2.0 implementations of C++, nested types were made visible to code outside the block in which the nested type was defined.

In 2.0 and later implementations of C++, nested types are not visible to code outside the block.

To accommodate prior practice, turn off toggle *Nested_types*. Turning the toggle Off causes any nested types to be declared outside the class as well as inside, in the first enclosing non-class scope. In this example, *T* would be declared within *s* as well as at the global scope, so both *s*: *T* (the nested-class syntax) and *T* alone (the old syntax) suffice to refer to *T*. See the section “C++ - Specific Issues” in Chapter 11, “Compiling and Linking C++.

New_call_frame

Default: On for -O3 and higher

Optimize call frame and epilog for Pentium efficiency.

Noalias

Default: Off

Cause optimizer to assume all variables are non-aliasable.

When toggle *Noalias* is On, the optimizer assumes that no variable, pointer-based or otherwise, can be aliased (that is, accessed or modified through any indirect means, such as through a pointer dereference), unless it is explicitly qualified with type qualifier *-Alias*.

WARNING: Turning On toggle *Noalias* for a program that has aliased pointers will cause the generated code to run incorrectly.

Optimize_for_space

Default: Off unless optimization level = s

When this setting is On, the code generator produces code sequences that minimize space, sometimes at the expense of execution speed (SCDE Release 2.01 or later).

Optimize_FP

Default: On

When this setting is On, function prologue and epilogue sequences that, respectively, set up and restore the local stack-frame register **ebp**, are omitted if **ebp** is not used. The only disadvantage of this optimization is that functions so optimized do not appear in any run-time stack dump, and their omission can further limit the stack-dump routine's ability to identify the active functions in the dump. When Off, Save pointers are generated for all functions (SCDE Release 2.01 or later).

Overload_info

Default: Off

C++ only: Show function chosen during overload operation.

If toggle *Overload_info* is turned On in C++ compilations, the compiler informs you which function was chosen during overload resolution. This can help if you are not sure at a particular point in your program which function (of a set of overloaded functions) is being called. The information is emitted as a warning.

Pack_text

Default: On with -Oc; otherwise Off

This setting inhibits alignment in the text segment (SCDE Release 2.01 or later).

Parm_warnings

Default: On

When this setting is On, the compiler produces warnings when arguments to a non-prototype (old-style) function F do not match the types of the declared formal parameters of F. Frequently this inconsistency is a source of disastrous or hard-to-find bugs.

Example:

```
double square(x) double x; {return x*x;}
...
printf("%d\n", square(3));
```

square is passed the integer 3, not double-precision 3.0, and the compiler issues a warning. The C language definition prohibits the compiler from casting 3 to a double before passing it.

To eliminate the compiler warnings, turn Off the toggle *Parm_warnings*. The program text should be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

PCC

Default: On (-Xt mode)

Many UNIX C programs that compile under the Portable C Compiler do not compile under an ANSI compiler such as High C. When the *PCC* toggle is turned On, many of the ANSI restrictions are relaxed so that such programs are more likely to compile (with appropriate warnings).

The following is a list of the ANSI restrictions that this toggle affects:

- Structure member selection is permitted on a variable of any type; it need not be a struct or union. However, if the name of the member being selected is declared as a member in more than one struct or union definition, each occurrence must be declared with the same type and be mapped at the same offset.
- Any pointer type is considered compatible with any other pointer type.
Note: Turning On the toggle *Pointers_compatible* has this effect.
- Pointers and any integer type are considered compatible.
Note: Turning On the toggle *Pointers_compatible_with_ints* has this effect.

The preprocessor is also put in a mode that more closely conforms to pre-ANSI C compilers. Macro arguments can appear between single quotes:

```
#define CTRL(x) ('x' & 037)
```

A comment between tokens can be used to concatenate them into a single token:

```
#define CAT(x,y) x/**/y
```

To use the unsignedness-preserving semantics employed by PCC, you must turn On the *Use_UP_Rules* toggle.

These PCC language features are not supported:

- (cast) variable as left side of assignment
- old-fashioned assignment operators (=+)
- old-fashioned initialization (=)

See also the **-Hpcc** compiler option and **-Xt** option in the Chapter 2, “Invoking the Compiler.”

Pointers_compatible

Default: Off

When this setting is On, the compiler permits a pointer value to be assigned to an incompatible pointer variable, with an appropriate warning.

Pointers_compatible_with_ints

Default: Off

When this setting is On, pointers of any type are allowed to be compatible with ints, with an appropriate warning. Although this is in violation of the Standard and High C specifications, many old C programs improperly assign pointers to ints and vice versa. This toggle allows such programs to be compiled without modification.

ANSI and High C disallow this dangerous practice because pointers are not necessarily the same size as ints on all machines.

Postfix_takes_two_args

Default: On

C++ only: Force overloaded postfix operators to require two arguments.

In C++ compilations, when you overload the postfix operators ++ and --, you must use two arguments to distinguish them.

Turn Off toggle *Postfix_takes_two_args* to eliminate this distinction. Then you can define just one ++ operator in a class, and invoke it either with the syntax ++x of the syntax x++.

Postpone_arg_pops

Default: On

This toggle provides an optimization to slightly reduce code space and increase execution speed. After returning from a function call whose arguments were pushed on the stack, code must be executed to “pop” the arguments by adding an appropriate amount to the stack pointer. If this toggle is On, the compiler might postpone the popping of arguments and combine it with popping arguments for a subsequently called function.

Note: This toggle has no effect if the code is being generated for an 80486, Pentium, or Pentium Pro (that is, the **-486**, **-586**, or **-686** command-line option was specified). Arguments are not pushed or popped from the stack in 486 mode.

Preload_args_from_memory

Default: Off unless optimization level = 1, s

When this setting is On, the compiler copies function arguments to machine registers in the function prologue to reduce memory fetches. See the “Level 1 Optimizations - Preloading Arguments from Memory” section in Appendix C, “Optimizing Program Performance”, for more information.

CAUTION: Do not turn on toggle *Preload_args_from_memory* when toggle *Global_CSE* is On.

Print_lattice_members

Default: Off

C++ only: Include a list of class members in the class lattice display.

Turn on toggle *Print_lattice_members* to include a list of class members along with the class-lattice display produced with toggle *Print_lattices*. The members are displayed graphically.

For example, given this program text:

```
class { int v; };
class A { int a; };
class B : A, virtual V { };
class C : A, virtual V { };
class D : B, C { void f ( ); };
```

this is the graphical display for class D:

```
D {f}
-----
B           C
-----
A {a} v:V {v} A {a} v:V {v}
```

See the following toggle *Print_lattices* for an explanation of this format.

Print_lattices

Default: Off

C++ only: Generate a graphical view of any class with one or more bases.

Use toggle *Print_lattices* to obtain a graphical view of any class that has one or more bases. The output is simple character text that can be displayed on any device.

The lattice is printed in two forms:

Firstform: The first form is a one-dimensional linear representation that uses brackets to enclose classes. For example, class *some_class* with bases **bl** ... **bn** is displayed as:

```
(some_class bl...    bn]
```

where **bl** ... **bn** is the display of the bases.

Second form: The second form is a graphical form. A class *some_class* with bases **bl** ... **bn** is displayed as this:

```
some_class
-----
bl    ...    bn
```

where **bl** ... **bn** is the display of the bases.

Virtual bases are replicated in the display and are preceded by v: to identify them as virtual bases.

For example, given this program text:

```
class V { int v; };
class A { int a; };
class B : A, virtual V { };
class C : A, virtual V { };
class D : B, C { void f( ); };
```

the compiler produces the following linear lattice display for class D:

```
Class lattice for D at test.cpp, L5/C1, in the
format [D Bl...    Bn]: [D [B A v:V] [C A v:V] ]
```

and this graphical lattice display, but with duplication of any virtual bases:

```
D
-----
B      C
-----
A v:V A v:V
```

See also toggle *Print_lattice_members*.

Print_layout

Default: Off

C++ only: Print layout of C++ data structures.

When toggle *Print_layout* is On, the compiler prints to standard output the layout of all C++ data structures in the module being compiled.

Print_protos

Default: Off

When this setting is On, the compiler writes to standard output a new, prototype-style function header for each function definition. This toggle aids in the conversion of C programs to the ANSI prototype syntax derived from the C++ language. For example, for the function definition:

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces the following output:

```
int f(int *x, double (*y)(), int z[]);
```

The old function header can then be replaced with the generated one.

Print_template_usage

Default: Off

C++ only: Display templates used in a C++ compilation.

Turn On toggle *Print_template_usage* to find out which templates were used in a C++ compilation. Both function templates and class templates are printed.

Print_var_info

Default: Off

Display mapping of auto-, register-, and argument-class variables.

When toggle *Print_var_info* is On, the compiler displays where each auto-, register-, or argument-class variable is mapped on a per-function basis.

One of the following formats is used:

<code>N ==> R</code>	Specifies variable N has been mapped to machine register R
<code>N+offset ==> R</code>	Specifies a member of struct N has been mapped to machine register R
<code>N ==> FP+offset</code>	Specifies variable N has been mapped into the stack frame at offset from base pointer FP
<code>N ==> <eliminated></code>	Indicates variable N was not referenced (or all references were optimized away); the variable was not mapped

Prototype_conversion_warn

Default: On if `Warning_Level > or = 2`; Off otherwise.

When this setting is On, the compiler generates a warning message when a function's argument is converted due to a prototype declaration.

When using function prototypes, the compiler may automatically convert a function's argument so that the argument's type matches that of the formal parameter. Wherever such a conversion does not match what would happen in the absence of prototypes, such C code would probably not run correctly on older C compilers that lack prototypes.

Prototype_override_warnings

Default: On

When this setting is On, the compiler produces a warning when a declaration (not definition) for a function using the new prototype syntax overrides the semantics of an old-style function definition.

Standard C requires that function prototype declarations override old-style function definitions. This means that the simple inclusion of a `.h` header file with prototype declarations of functions obtains the new prototype semantics for the definitions of those functions. This feature has both disadvantages and advantages.

The advantage is that the new prototype semantics—the Pascal-style assignment-conversion of arguments to the types of the formal parameters—is obtainable by merely including a declaration in a header file. The disadvantage is that a definition can no longer be read out of context; without searching header files one cannot determine whether or not the compiler compiles the function using prototype-style semantics.

Example:

```
file header.h:
int func(float f, long l);
file prog.c:
#include "header.h"
int func(f, l) float f; long l; {
    ...
}
void sub() {
    func(3, 4.4); /* Passes 3.0 and 4L via */
} /* automatic conversion. */
```

Were **header.h** not included, the call to `func` in `sub` would pass the `int` 3 and the double 4.4, and probably `func` would not work properly. With the header file included, the interface for `func` is changed to prototype-style (3 is converted to `float` and 4.4 to `long`). Thus, one can find out how the compiler treats `func` only by searching all the header files.

To obviate the need for searching, High C provides a warning message whenever an old-style definition is overridden by a prototype. The warning message can be disabled by turning Off toggle *Prototype_override_warnings*.

Function definitions should be written with the new prototype syntax for improved readability and reliability. For example:

```
file prog.c:
int func(float f, long l) {
    ...
}
```

The ANSI committee permitted the override feature for two reasons: First, it would take some work to convert programs to use the new syntax in the definition (although with toggle *Print_protos*, High C generates the headers from old-style definitions); Second, most compilers do not support prototype-form definitions, and the use of a header that is conditionally included based upon the compiler being used makes code more easily compilable by different compilers.

Read_only_strings

Default: Off

When this setting is On, string constants are unconditionally placed in a read-only data section. Two or more identical string literals in the same module appear only once in memory. It is an error to modify the storage of a string constant at run time when the toggle is On.

When the toggle is Off, only those string literals being cast to `const char *` (implicitly or explicitly) are placed in a read-only section.

Recognize_library

Default: On

When this setting is On, the compiler may substitute in-line code for any function defined in the ANSI Standard C library. Candidates for substitution currently include the math and string functions. Future releases may encompass more of the library.

Recognize_setjmp_longjmp

Default: Off

Generate code directly for `setjmp( )` and `longjmp( )`.

When toggle *Recognize_setjmp_longjmp* is On, the compiler generates code directly for `setjmp( )` and `longjmp( )` instead of making calls to the run-time library (which can be costly for applications that are `setjmp( )` and `longjmp( )` intensive).

Generating code directly always reduces the number of instructions necessary to do a `setjmp()` or a `longjmp()`; it might also make code larger.

Note 1: Toggle *Recognize_setjmp_longjmp* has no effect unless toggle *Recognize_library* is on.

Note 2: This should not be used if ABI object compatibility is an issue.

Reduce_reg_contention

Default: Off unless optimization level ≥ 2

When this setting is On, the register allocator attempts to limit the lifetime of registers to reduce spill code. See the “Level 2 Optimization–Spill Analysis” section in Appendix C, “Optimizing Program Performance”, for more information (SCDE Release 2.01 or later).

Reg_alloc_enhance

Default: Off unless optimization level ≥ 2

When this setting is On, the compiler attempts to improve register allocation. See the “Level 2 Optimization - Enhanced Register Allocation” section in Appendix C, “Optimizing Program Performance”, for more information (SCDE Release 2.01 or later).

Reorder_scratch_regs

Default:

- Off if optimization level = 0 - 6, or s, or if you specify option - **g**
- On if optimization level = 7

Improve instruction scheduling.

When toggle *Reorder_scratch_regs* is On, the compiler performs some added analyses to improve instruction scheduling.

See the section “Instruction Scheduling” in Appendix C, “Optimizing Program Performance”, for more information.

Resolve_branch_targets

Default: On with -486

This specifically aligns branch targets, for instance, the destination of branches/calls/returns to avoid i486 pipeline stalls. Padding with nops will be done to avoid cases where a target instruction spans two i486 cache lines. This is made the default if -486 and optimization levels of >2 are used. Entry points are aligned on a 16-byte boundary. Note that code expansion is limited and the execution speed increases in most cases on the i486 processor (SCDE Release 2.01 and later only).

Note that for all optimization levels except -Os, -Oc, and -486 with levels over 2, the following toggles are enabled:

Align_labels, Align_routines

with alignment to 4.

For -Oc, Pack-text is enabled (SCDE Release 2.01 or later).

RTTI_comdat

Default: On

C++ only: Solve the single-copy problem for RTTI structures (COMDAT).

When toggle *RTTI_comdat* is On, it solves the single-copy problem for classes without seats declared within its scope, or for non-class types.

For a discussion of the single-copy problem and its solutions, see the section “The Single-Copy Problem” in Chapter 11, “Compiling and Linking C++”.

When you use option -Hnocomdat, this toggle is disabled (turned Off).

Run_time_type_info

Default: Off

C++ only: Generate and store run-time type information (RTTI) for a class.

Toggle *Run_time_type_info* ensures that, for every class compiled within its scope, run-time type information (RTTI) is generated and stored in the class's virtual function tables.

If toggle *Run_time_type_info* is not turned On, RTTI is generated only as needed by typeid, and there is no guarantee that the virtual-function tables will contain a pointer to RTTI.

Source_code_order

Default: Off unless optimization level = 0, 1, -g, ql, or -qb

When this setting is On, the compiler attempts to produce code in source order. This makes your code easier to debug at the assembly language level.

Note: This toggle is turned On when option **-g** is specified.

Strength_reduce

Default: Off unless optimization level ≥ 3

When this setting is On, the compiler performs loop strength reduction. See the “Level 5 Optimization–Loop Strength Reduction” section in Appendix C, “Optimizing Program Performance”, for more information.

Strict_ANSI_math

Default: On

When this setting is On, the compiler does not generate in-line code for transcendental functions that set errno to indicate existing error conditions. For a program to be strictly ANSI conforming, this toggle must be turned On.

Tail_recursion

Default: On

When this setting is On, the compiler attempts to optimize recursive functions. See the “Additional Ways to Optimize Performance - Tail Recursion” section in Appendix C, “Optimizing Program Performance”, for more information.

Template_trivial_conversions

Default: Off

C++ only: Allow trivial conversions for matching arguments to function templates.

In C++ compilations, turn On toggle *Template_trivial_conversions* to permit so-called trivial conversions (as defined in the **Annotated C++ Reference Manual**) to be allowed when matching arguments to function templates.

For more information, see the section entitled “Problems with Matching const Parameters” in Chapter 11, “Compiling and Linking C++”.

Treat_exports_as_locals

Default Off

Change public functions to static functions.

When toggle *Treat_exports_as_locals* is On, the compiler changes public functions to local static functions. This toggle tells the compiler that all of the modules needed by the program are contained within the current source module and the only public entry point is function `main()`.

Toggle *Treat_exports_as_locals* also tells the compiler that any non-local function is contained within a library.

Trigraphs

Default: Off unless you specify option **-Hansi**

Enable trigraphs.

When toggle *Trigraphs* is On, the compiler recognizes trigraph sequences of three characters that are converted to a single character. Trigraphs are useful on machines that have limited character sets; they allow you to represent characters that would otherwise be unavailable.

On ASCII-based systems trigraphs are not generally useful. All the characters that trigraphs represent are typically available on contemporary keyboards.

Trigraphs are required for ANSI Standard C compatibility, so option **-Hansi** turns On toggle *Trigraphs*.

Try_static_cast

Default: Off

C++ only: Replace old-style type casts with static - cast.

Toggle *Try_static_cast* causes all old-style C++ casts within its scope of the form *(Type) expression* to be treated as the new type cast *static-cast*. This toggle also causes the compiler to issue a warning at each line containing a cast.

unordered_compares

Default: Off

Make floating-point compares with NaNs conform to IEEE Standard 754.

When toggle *unordered_compares* is On, floating-point compare operations are generated so that comparisons with NaNs (not-a-number) conform to IEEE Standard 754. The overhead for such conformance is typically an extra instruction for each floating-point compare.

Use_BSS

Default: On

Map uninitialized static variables to offsets in the BSS section.

Uninitialized static variables are ordinarily mapped to offsets in a so-called BSS section named `.bss`. When toggle *Use_BSS* is Off, such variables are placed in the default data segment instead.

Use_frame_pointer

Default: Off for -486, -586, and -686

This toggle can be used in conjunction with the *486* toggle to improve execution speed. When Off, the compiler uses the stack-pointer register `esp` to reference items in the stack frame, instead of the `ebp` register. The `ebp` register is then freed up to be used for other purposes. This optimization is suppressed for any function that invokes the `alloca()` function. It is also suppressed if `-g` is specified.

Execution speed is improved for two reasons: the overhead of setting up the `ebp` register is omitted; and an extra register is available to the register allocator, which increases the number of variables and intermediate computations that can be retained in registers.

There is a drawback to turning Off this toggle. Debuggers typically use the content of each instance of the `ebp` register as a linked list from which a call trace can be displayed. Such a facility will not work for code that has been compiled with this toggle Off. This also causes C++ exception handling to fail, and should not be used with C++ compiles.

Use_reciprocal_for_divide

Default: Off

When this setting is On, a divide by a constant will be converted to a multiply by the reciprocal of the constant, because a multiply is typically faster than a divide.

CAUTION: This can introduce inaccuracy into a program because the reciprocal may not be exact. In general, the last bit of the result might be different. For most programs, this does not matter; the toggle is provided for users who want the extra performance.

Note: Even if this toggle is Off, the compiler might choose to multiply by the reciprocal if the reciprocal has an exact binary representation; for example, the reciprocal of 2.

Use_UP_rules

Default: Off

In C there are two rules for implicitly widening an unsigned char or unsigned short operand in an arithmetic expression: value-preserving (VP) and unsignedness-preserving (UP).

The VP rule widens an unsigned char or unsigned short to signed int. This is the rule used in Standard C.

The UP rule widens such operands to unsigned int. This rule is used by older compilers, such as those based on the AT&T Portable C Compiler.

Some programs can behave differently depending on which rule is in force. For example:

```
unsigned char c;
...
if (c-1 > 0) foo();
```

If `c` happens to have the value zero when the `if` statement is executed, the relational expression is false if the `VP` rule is used, but true if the `UP` rule is used. This latter characteristic occurs because `c` is widened to an unsigned int; therefore, the type of the expression `c-1` is also unsigned int.

If you are compiling code that was originally developed under a pre-ANSI compiler, turn On *toggle Use_UP_rules*.

See also the description of the toggle, *VP_UP_warn*

Default: Off if `Warning_Level < 4`; On otherwise.

This toggle is enabled for transition (that is, **-Xt**) compile mode.

VP_UP_warn

Default: Off if `Warning_Level < 4`; On otherwise.

When this setting is On, the compiler issues a warning when it encounters an expression that could have a different value depending on whether value-preserving or unsignedness-preserving rules are used. See the description of the *Use_UP_rules* toggle. This toggle is enabled for transition (that is, **-Xt**) compile mode.

Vtable_common

Default: On

C++ *only*: Solve the single-copy problem for virtual function tables (COMDAT).

When toggle *Vtable_common* is On, it solves the single-copy problem for virtual function tables, for classes without a seat declared within the scope of the toggle.

For a discussion of the single-copy problem and its solutions, see the sections “Using Templates” and “Instantiating Templates Manually” in Chapter 11, “Compiling and Linking C++”

Widen_float_args

Default: On unless option **-fsingle2** is specified

An argument of type `float` passed to a non-prototyped function, or passed as a variable argument to a prototyped function, is ordinarily widened to `double` before being passed. This is the Standard C convention.

When this toggle is turned Off, no such conversions are performed.

CAUTION: Beware of passing a `float` argument to a function that was compiled with a different setting for this toggle. For example, when this toggle is Off, the values of `float` expressions cannot be displayed with `printf()`.

Data Representation and Storage Mapping

This chapter provides certain information required by the ANSI C Standard. It describes the following:

- Implementation Defined Behavior
- Storage Mapping for C++ Class Objects

Implementation Defined Behavior

The ANSI C Standard (ANSI document X3.159) requires that each C compiler implementation provide information about how various "system specifics" behave. The following sections document this implementation-defined behavior for High C/C++.

Identifiers

Number of Characters in an Identifier

The compiler places no limit on the number of characters in an identifier, with or without external linkage.

Identifier Case-Sensitivity

External identifiers are case-sensitive.

The implementation used to map sequences of characters to external source file names is case-sensitive.

Number of Identifiers

The compiler places no limit on the number of external identifiers in one translation unit, the number of identifiers with block scope declared within one block, or the number of macro identifiers simultaneously defined in one translation unit.

Characters

Source and Execution Character Sets, Members and Mapping Members

Members Source and execution character sets are both standard ASCII. Mapping Each character in the source character set maps into the identical character in the execution character set. Without exception, all character constants map into some value in the execution character set.

Bits in a Character

A character is stored in a byte (eight bits).

'Char' Type Compatibility

The type specifier `char`, when not accompanied by an adjective, denotes by default an unsigned character type. You can change the default type by turning Off the toggle *Char_default_unsigned*.

However, even though you make the `char` type be signed or unsigned, `char` still is not type-compatible with signed `char` or unsigned `char`. To make `char` be type-compatible with whichever type is the default, turn On the toggle *Char_is_rep*. This toggle causes the `char` type to be identically either signed `char` or unsigned `char`, depending on the default.

See the discussions of *Char_default_unsigned* and *Char_is_rep* in Chapter 4, “Compiler Toggles” for clarification.

Character Constants

In C, a character constant is of type **int**. The ANSI C Standard requires the type to remain `int`, even if truncation occurs.

Multi-Character Constants

High C/C++ permits a character constant that contains more than one character. This is the value of a two-character constant:

$(256 * (\text{value of the first character})) + (\text{value of the second character})$

For example, the value of 'ab' is $(256 * 'a') + 'b'$.

Wide-Character Constant of Multi-Byte Characters

The value of a wide-character constant that contains N multi-byte characters is the value of the last (Nth) multi-byte character.

For example, the value of L'abcd' and of L'cd' is the value of L'd'.

Integers

Integer Representations

64-Bit Integer Datatype

A new data type has been added to the compiler to enable use of 64-bit integers. Specifying the type long long and unsigned long long creates variables that are signed and unsigned 64-bit integers. As the use of these variables is an extension to the data types provided by ANSI C, code that uses them will not compile for the strictly conforming ANSI C compile modes when -Xc or -Hansi flags are specified.

There are no restrictions on the use of 64-bit integer variables. They can be used in any context where long and unsigned long variables are used. Promotion rules are the same as those followed when converting from smaller integral types to long or unsigned long.

Constants that are to have the 64-bit integer data types must have a suffix appended to indicate that the constant type is not the standard integer type. For example, the constants 3LL and 3ULL have the types long long and unsigned long long, while 3 has the type integer. Care should be taken when passing constants to unprototyped functions that expect 64-bit data type values; omitting the constant type suffix causes the constant to be passed as a 32-bit value, not the 64-bit value. Integral constants are not automatically promoted to a 64-bit data type when a constant overflows the long or unsigned long data types. The appropriate suffix must be used if the constants are to be represented as 64-bit integers.

Support has been added to the C library to permit the conversion to and from string integer constants and 64-bit data variables using any of the `printf(g3s)` or `scanf(g3s)` family of functions. The symbols `LONG_LONG_MIN`, `LONG_LONG_MAX`, and `ULONG_LONG_MAX` are defined in `<limits.h>` to define the signed minimum, signed maximum, and unsigned maximum values that can be represented by the 64bit data types.

Integers are represented in twos-complement binary form; there are four bytes in an **int**. Various integer types are restricted to certain ranges of values, as shown in the table below.

Table 5-1. Integer Data Types

Integer Data Type	Size Bytes	Align-ment	Minimum	Maximum
signed char	1	1	-128	127
unsigned char	1	1	0	255
short	2	2	-32,768	32,767
unsigned short	2	2	0	65,535
int	4	4	-2,147,483,648	2,147,483,647
unsigned int	4	4	0	4,294,967,295
long	4	4	-2,147,483,648	2,147,483,647
unsigned long	4	4	0	4,294,967,295
long long	8	4	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	8	4	0	18,446,744,073,709,551,614

Integer Conversions

Integer/Float Conversions

These are the rules for conversions in which the result cannot be represented exactly:

int to float: convert to the nearest value,

higher or lower float to int: round toward 0 (zero)

Converting an Integer to a Shorter Signed Integer

An integer is converted to a shorter signed integer or int bit field by bit truncation. That is, when an X-bit value Xval is to be stored into a Y-bit receptacle Yval, where $X > Y$, the rightmost Y bits of Xval are stored in Yval.

Converting an Unsigned Integer to a Signed Integer

An unsigned integer U is converted to a signed integer I where `sizeof(U) = sizeof(I)` by transferring the bits of U into I, whether or not the value of U is representable in I.

Note: If the value of an unsigned integer is larger than the largest positive signed integer value, it is interpreted bit by bit and yields a negative value.

Operations Involving Signed Integers

Integer Division with Negative Operands

In an integer divide operation (X/Y), when either of the operands is negative, the quotient is truncated toward 0 (zero).

In an integer remainder operation ($X \% Y$), when either or both of the operands is negative, the following is true:

$X \% Y$ is the same as $X - (X/Y) * Y$

Bitwise Operation

The result of a bitwise operation on a signed integer is the same as if the integer were treated as unsigned.

Sign of the Remainder

The sign of the remainder on integer division is the same as the sign of the dividend.

Right Shift of a Signed Integer

The right shift of a signed integral type is arithmetic; that is, the sign bit is propagated to the right.

Floating-Point Numbers

Floating-Point Representations

High C/C++ uses the IEEE Standard 754 formats to represent floating-point data. Each floating-point data type has a sign bit, exponent bits, and mantissa bits. The following table summarizes this information.

Table 5-2. Floating Point Data Types

Floating-Point Data Type	float	double	long double
Size (Bytes)	4	8	12
Alignment (Bytes)	4	4	4
Sign Bit	1	1	1
Exponent (Bits)	8	11	15
Mantissa (Bits)	23	52	64
Minimum Value	8.43×10^{-37}	4.19×10^{-307}	3.362×10^{-4932}
Maximum Value	$3.37 \times 10^{+38}$	$1.67 \times 10^{+308}$	$1.890 \times 10^{+4932}$

For details about the IEEE data format, refer to the *ANSI/IEEE Standard 754/1985* documents.

Floating-Point Conversions

The default rounding mode is "round to nearest". These are the rules for conversions in which the result cannot be represented exactly:

float-to-int Conversions

When a floating-point number is truncated to an integral type, the truncation is toward 0 (zero). For example:

- 2.7 is truncated to -2
- 1.2 is truncated to -1
- 2.7 is truncated to 2

int-to-float Conversions

When an integral type is converted to a floating-point number, the truncation is toward the nearest value, higher or lower.

double-to-float Conversions

When a double-precision value is converted to a floating-point number, the following rounding-mode rules apply:

- For machine arithmetic, round to the nearest lower value.
- For constants, round to the nearest value, higher or lower.

Arrays and Pointers

Arrays

Arrays are stored in row-major order. This means that the last subscript in a multi-dimensional array varies fastest. For example, this array:

```
int x[4][6]
```

is stored as this:

```
x[0][0], x[0][1], x[0][2], ..., x[3][3], x[3][4], x[3][5]
```


Type of `size_t`

The type of `size_t`, the value returned by `sizeof( )` (that is, the type of integer required to hold the maximum size of an array) is type **int** if representable as an **int**; otherwise, it is type **unsigned**.

Pointers

Casting a Pointer to an Integer

For information about how the High C/C++ compiler handles a pointer cast to an **int** type, see toggle *Pointers_compatible_with_ints* in Chapter 4, “Compiler Toggles.”

When toggle *Pointers_compatible_with_ints* is On, the compiler allows pointers of any type to be compatible with int types, with an appropriate warning. This is the default behavior.

Difference of Two Pointers

The difference of two pointers is of type *ptrdiff_t*, which is defined as type 'int' in *stddef.h*. Using Keyword register to Place Objects in Registers:

- An auto- or register-class variable can spend a portion or all of its lifetime in a machine register or even in multiple registers.
- Multiple variables can be mapped to a single register.

The compiler attempts to map each auto- or register-class variable to a register if its type is appropriate and its address is not taken. Auto variables that are not mapped into registers are placed at fixed offsets within the function's stack frame. This is necessary if the variable's address is required, or if the variable is of an aggregate type (for example, struct or array).

A register variable is treated exactly as an auto variable insofar as code generation is concerned, except when a function calls `setjmp( )`. In such a case, all auto variables are forced to memory so that their values are not lost across a subsequent `longjmp( )` call.

Using Keyword `register` to Place Objects in Registers

An `auto`- or `register`- class variable can spend a portion or all of its lifetime in a machine register or even in multiple registers. Multiple variables can be mapped to a single register.

The compiler attempts to map each `auto`- or `register`- class variable to a register if its type is appropriate and its address is not taken.

`auto` variables that are not mapped into registers are placed at fixed offsets within the function's stack frame. This is necessary if the variable's address is required, or if the variable is of an aggregate type (for example, struct or array).

A `register` variable is treated exactly as an `auto` variable insofar as code generation is concerned, except when a function calls `setjmp()`. In such a case, all `auto` variables are forced to memory so that their values are not lost across a subsequent `longjmp()` call.

This strategy changes for `-g` code generation, as the debuggers expect variables to always be mapped to one storage location.

Note that more than one variable or a compiler temporary may share the use of a register, if they are not used together for a part of a function.

How the Compiler Maps Variables to Storage

The compiler defines up to six default control sections for placing code and data, in addition to any you might define with `pragma Code` or `pragma Data`. The following table summarizes this.

Table 5-3. Default Control Sections for Code and Data

Name	Class Contents
<code>.bss</code>	Data Uninitialized variables (that are not <code>const</code> qualified)
<code>.data</code>	Data Initialized static variables (local and global) and writable strings referenced from code
<code>.data1</code>	Data Writable strings whose addresses are referenced from within the <code>.data</code> section

Name	Class Contents
.rodata	Literal Read-only string constants and read-only static variables (that is, those declared with the const qualifier)
.text	Text Executable code

Local Static Variables

Unless specified otherwise with a pragma **Data** specification, each local static variable is placed in one of the following sections:

.rodata, if const qualified

.data, if initialized

.bss, if uninitialized

Uninitialized Global Variables

Each uninitialized global variable with no extern specifier is defined as a common block or placed in the .bss section (depending on the state of the toggle *Multiple_var_defs*).

Initialized Global Variables

Each initialized global variable with no extern specifier is mapped into the .data section and given the global attribute.

Auto Variables

Each auto variable is assigned either to a machine register or to storage in the routine's stack frame. The compiler attempts to place each auto-classed variable in a register, provided the variable's type is appropriate and its address is not required.

If `setjmp()` is called within a function, all auto variables are mapped to memory unless the register class is explicitly specified, so that the semantics of `setjmp()/longjmp()` work correctly.

Extern Variables

Each extern variable is given the global and undefined attributes. Const-qualified variables:

Each const-qualified static initialized variable, whether local or global, is mapped to the read-only data section. For example:

```
main ()
{
    static const char hextable[] = "0123456789abcdef";
    int i;
    ...
    f(hextable[i]);
}
```

Any attempt to directly assign to hextable elicits an error message from the compiler. Any indirect attempt, such as through passing a pointer to A or to hextable, will cause a protection fault or other error at execution time.

Note that string literals are placed in a read-only section only if they are part of a const data object. For example:

```
const char * P = "a string";
```

Here "a string" is placed in the read-only section, but P is placed in data. To place P in the read-only section, do this:

```
const char * const P = "a string";
```

Structures, Unions, Enumerations, and Bit Fields

Size, Padding, and Alignment of Structures, Classes, and Unions

The size of a struct, class, or union depends on whether the compiler generates padding to align the construct's members. The compiler generates such padding by default under the control of the **Align_members** pragma.

Keywords **_Packed** and **_Unpacked**

The keywords **_Packed** and **_Unpacked** have been added to High C/C++ to allow control over member alignment on an individual struct, class, or union basis. A **_Packed** struct or class is not padded; an **_Unpacked** struct or class is padded.

CAUTION: A struct, class, or union is aligned according to the most stringent requirements of its members. However, members of a **_Packed** struct or class might not be aligned on word boundaries.

Unpadded union

The size of an unpadded union is the size of its largest member. Padded union:

The size of a padded union is the size of its largest member, padded so that its size is evenly divisible by its alignment.

Unpadded struct or class

The size of an unpadded struct or class is the sum of the sizes of its members. Non-bit-field members always start on byte boundaries and there is no padding between members, except in bit members.

Padded struct or class

The size of a padded struct or class is the sum of the sizes of its members, including alignment padding between members. A padded struct or class is padded so that its size is evenly divisible by its alignment.

C++ only struct, class, or union

A struct, class, or union containing elements specific to C++ might contain compiler-defined members that increase the size. See “Storage Mapping for C++ Class Objects” later in this chapter for more information.

Size of Enumerations

The size of an enum type depends on the status of the *Long_enums* toggle (see Chapter 4, “Compiler Toggles”).

- If toggle *Long_enums* is Off, the enum type maps to the smallest of one, two, or four bytes, such that all values can be represented.
- If toggle *Long_enums* is On (default), an enum maps to four bytes (required for Intel ABI conformance).

Bit Fields

Plain int versus unsigned int

High C/C++ supports signed and unsigned bit fields.

- A bit field declared as int is treated as unsigned int.
- A bit field declared as signed int is interpreted as signed.

By default, bit fields are unsigned - that is, int x:11 specifies an unsigned bit field, and long x:19 specifies an unsigned long bit field. You can use the keywords signed and unsigned to specify signedness and unsignedness, respectively.

Bit-field Widths

A bit field cannot exceed 64 bits and is packed in consecutive bytes.

The following table shows the possible widths for bit fields, where w = maximum width (in bits).

Table 5-4. Bit-Field Types

Bit-Field Type	Max. Width w (Bits)	Range of Values
signed char	1 to 8	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
char	1 to 8	0 to $2^w - 1$
unsigned char	1 to 8	0 to $2^w - 1$
signed short	1 to 16	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
short	1 to 16	0 to $2^w - 1$
unsigned short	1 to 16	0 to $2^w - 1$
signed int	1 to 32	$-2^{(w-1)}$ to $2^{(w-1)} - 1$

Bit-Field Type	Max. Width w (Bits)	Range of Values
int	1 to 32	0 to $2^w - 1$
unsigned int	1 to 32	0 to $2^w - 1$
signed long	1 to 32	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
long	1 to 32	0 to $2^w - 1$
unsigned long	1 to 32	0 to $2^w - 1$
signed long long	1 to 64	$-2^{(w-1)}$ to $2^{(w-1)} - 1$
long long	1 to 64	0 to $2^w - 1$
unsigned long long	1 to 64	0 to $2^w - 1$

Storage-unit Boundaries for Bit Fields

A storage-unit boundary for a bit field is determined by the number of bytes needed to store a value for the associated type.

- char bit fields have boundaries on every byte.
- short bit fields have boundaries on even addresses.
- long bit fields have boundaries on addresses divisible by 4
- long long bit fields are an exception, because they also have boundaries on addresses divisible by 4.

Bit-field Order of Allocation

The compiler allocates bit fields from the least significant bit to the most significant bit.

How the Compiler Maps Bit Fields

The compiler maps bit fields so that they never cross a storage-unit boundary. A bit field must fit within a four-byte word that is aligned to a four-byte boundary. If necessary, the bit field starts on the following unit boundary; padding is inserted where appropriate.

Bit fields affect the boundary alignment of the enclosing struct or union as follows:

- A field that does not cross a byte boundary imposes no alignment requirement on the struct.

- A field that crosses a single byte boundary requires the enclosing struct to be two-byte aligned.
- A field that crosses two or more byte boundaries requires the enclosing struct to be four-byte aligned.

Named Bit Fields

Named bit fields impose the same alignment on the enclosing struct or union as non-bit-field members do.

Unnamed Bit Fields of Non-zero Length

Unnamed bit fields of non-zero length do not affect the external alignment. In all other respects, they behave the same as named bit fields.

Unnamed Bit Fields of Zero Length

An unnamed bit field of zero length causes alignment to occur at the next unit boundary, based on its type. Thus, a zero-length bit field of type:

- `char` is aligned on the next four-byte boundary.
- `short int` is aligned on the next four-byte boundary.
- `int` is aligned on the next four-byte boundary.

An unnamed bit field of zero length also imposes a four-byte alignment on an enclosing struct. For example, this struct:

```
struct {  
    unsigned x:11, y:9, :0, z:13, w:1;  
    char c;  
    short i;  
} zed;
```

is mapped as:


```

zed+3 zed+2 zed+1 zed+0
332222222221111111111000000000
1098765432109876543210987654321
      yyyyyyyyyxxxxxxxxxxx
zed+7 zed+6 zed+5 zed+4
332222222221111111111000000000
1098765432109876543210987654321
      ccccccc wzzzzzzzzzzzz
zed+11 zed+10 zed+9 zed+8
332222222221111111111000000000
1098765432109876543210987654321
      iiiiiiiiiiiiiiiiii

```

This struct is aligned on address boundaries divisible by 4 because it contains the type. Note that the unnamed bit field forces padding, while alignment rules sometimes pad.

If `z` were changed to a `char` type, it would still be forced to begin in byte 4. If there were no unnamed bit field, `z` would begin in byte 2, 3, or 4, depending on whether it could fit in the space remaining without crossing its storage-unit boundary (which is 4).

Accessing a Volatile Object

When a data variable is qualified as volatile, the compiler must scale back optimizations normally applicable to data variables, as the volatile qualifier asserts that the code must not be changed with respect to:

- Order of memory accesses of volatile data variables
- The data variable must always be read or written from memory when it is referenced in the source
- The size and alignment of the data memory read or write must match the references in the source

This is to preserve the integrity of memory mapped hardware operations, or to stop optimizations that depend on the data variable not being changed for the scope of the operation.

Maximum Number of Declarators

The High C/C++ compiler places no upper limit on the number of pointer, array, or function declarators, in any combination, modifying an arithmetic, structure, union, or incomplete type in a declaration.

Maximum case Values in a switch Statement

The High C/C++ compiler places no upper limit on the number of case labels for a switch statement, excluding those for any nested switch statement.

Preprocessing Directives

Single-Character Constants in Conditional Preprocessing Directives

In a preprocessing directive, a single-character constant in a constant expression controlling conditional inclusion is always non-negative in value, ranging from 0 (zero) to 255.

Includable Source Files

Locating #include Files

By default, the compiler searches the standard directories for #include files. See descriptions of the -I<dir> and the -YI,<dir> commands in "Command Line Options" in Chapter 2, "Invoking the Compiler".

Quoted Names for #include Files

High C/C++ supports #include "filename.ext" and #include <filename.ext>.

Mapping of Source-File Character Sequences

High C/C++ does not map source-file character sequences.

The compiler expects ASCII characters.

Behavior on Encountering Pragma Directives

When the High C/C++ compiler encounters a supported pragma directive, it behaves as described for that pragma in Chapter 3, "Compiler Pragas."

When the compiler encounters an unsupported pragma, it emits a warning message. To suppress warning messages about an unsupported pragma, specify:

```
#pragma Push_ignore(Unsupported_pragma_name)
```

Compiler Limits

Items with No Limits

In compliance with the ANSI C Standard, the High C/C++ compiler places no limits on numbers of any of the following:

- Nesting levels:
 - compound-statement nesting levels
 - conditional-inclusion nesting levels
 - nesting levels of parenthesized declarators within a full declarator
 - nesting levels of parenthesized expressions within a full expression
 - levels of nested structure or union definitions in a single struct-declaration list
- Declarators:
 - - pointer, array, and function declarators, in any combination, modifying an arithmetic, structure, union, or incomplete type in a declaration
- Characters:
 - significant initial characters in an internal identifier or macro name
 - significant initial characters in an internal or external identifier
 - characters in a logical source line

- characters in a character string literal, or a wide string literal after concatenation
- Number of identifiers:
 - external identifiers in one translation unit
 - identifiers with block scope declared within one block
 - macro identifiers simultaneously defined in one translation unit
- Parameters:
 - parameters in one function definition
 - parameters in one macro definition
- Arguments:
 - arguments in one function call
 - arguments in one macro invocation
- Other components:
 - bytes in an object (hosted environment only)
 - case labels for a switch statement, excluding those for any nested switch statement
 - members in a single structure or union
 - enumeration constants in a single enumeration

Items with Limits

- `#include` nesting levels:
 - The limit for the number of nesting levels for files included with `#include` is 40. This is an arbitrary limit to detect recursive `#include` directives.

Storage Mapping for C++ Class Objects

This section describes how storage is arranged for class objects, treating such issues as virtual-function tables, virtual and non-virtual bases, and non-static members. You do not need this information to use High C/C++; it is provided for programmers who are interested in the layout of objects in storage.

Virtual-Function Tables

A class that contains virtual functions must have a virtual-function table, which is accessed whenever your program calls a virtual function. Each instance of a class with virtual functions contains a pointer to its virtual-function table. For example, if `fnc()` is a virtual function in `class x`, then `x.fnc()` essentially becomes `(x.vtab[index_of_fnc])()`, where `index_of_fnc` is the index of the function's address in the virtual-function table.

To avoid having multiple copies of the virtual-function tables for a class, the compiler produces a single copy of the table in the compilation unit that contains a function definition for the first non-pure-virtual, non-inline function member in a class.

In the following example, the virtual-function table is defined in the code generated for 2.cpp, and referenced from 1.cpp, in the call to `x.f`:

```
File class.h:
struct s {
    virtual f();
};
File 1.cpp:
#include "class.h"
s x;
int i = x.f();
File 2.cpp:
#include "class.h"
int s::f() {
    /* Body of f. */
}
```

If a class's member functions are all pure virtual or inline, the compiler makes a copy of the class's virtual-function table for each compilation unit that makes use of the virtual-function table. The compiler issues a warning for each such class. You can turn the warning off with pragma **Offwarn()** (see Chapter 3, "Compiler Pragmas."). We recommend that you not make a class's destructor inline if all other functions in the class are pure virtual or inline, because the virtual tables will then be generated in the module in which you define the destructor.

See also toggles *Import_vtabs* and *Export_vtabs* in Chapter 4, "Compiler Toggles".

Direct and Indirect Bases

A direct base of a class is another class listed in the first class's declaration; for example, in:

```
class X: Y,Z {...}
```

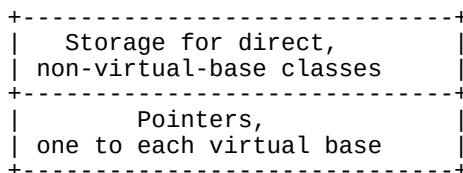
classes Y and Z are direct bases of X. An indirect base of a class X is a class that is not a base of X but is a base of one of X's direct bases; for example, if we add: `class A: X {...}` then class X is a direct base of A, and Y and Z are indirect bases of A.

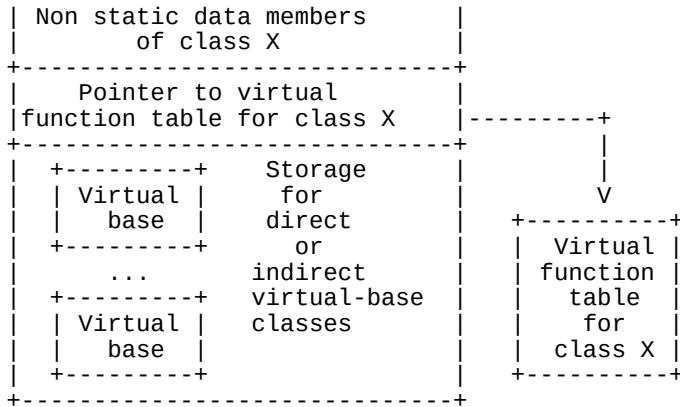
Suppose a class X has the following base classes:

- direct non-virtual-base classes
- direct virtual-base classes
- indirect virtual-base classes

Storage for the components of an instance of an object of class X with two virtual base classes is organized as shown in the figure below, except that redundant pointers might be optimized away.

Figure 5-1. Storage Organization for Object Instance with 2 Virtual Base Classes





The non-static data members are allocated the same way as struct members in plain C; that is, consecutively as they appear in the source, subject to boundary alignment. Access specifiers do not affect the mapping.

The direct, non-virtual bases laid out at the beginning of class X's storage do not include those bases' own virtual bases (if any); the storage for all virtual bases is in class X and not in any of its bases. X has a pointer to each of its direct virtual bases. When X is constructed, these pointers are initialized; so, too, are any virtual-base pointers that reside in X's direct or indirect bases (virtual or not).

So the storage for any base B allocated in an instance of X does not include any virtual bases of B. The storage for B in X may be thought of as a "truncated instance" of B.

If possible, rather than allocating a pointer to a virtual-function table within X, the compiler uses that same pointer in the first direct, non-virtual-base class B - provided B also has virtual functions.

The compiler rearranges the storage of direct, non-virtual-base classes so that the first base stored has a virtual-function pointer, if any such base does.

Consider this example:
`class A: B, X, D;`

In this case, if X has virtual functions and B does not, class X is stored first, followed by B and then D.

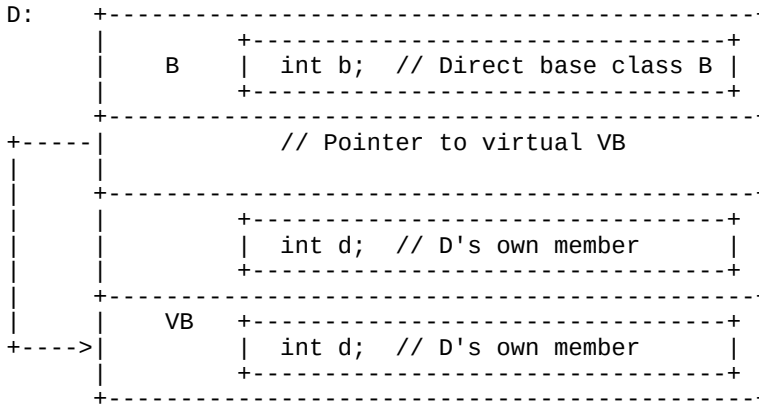
Here are two examples of class layout. The first one is reasonably simple:

```
class VB { int vb; };
```

```
class B { int b; };
class D: B, virtual VB { int d; };
```

The following figure shows the layout for an instance of class D:

Figure 5-2. Example of Class Layout



The next example involves virtual bases that themselves have virtual bases:

```
class VA { int va; };
class VB { int vb; };
class DVA: virtual VA {
    int dva;
}; // From virtual VA
class DVB: virtual VB {
    int dvb;
}; // From virtual VB
class DVB2: virtual VB {
    int dvb2;
}; // From virtual VB
class D: virtual DVA, virtual DVB,
    virtual VB, DVB2 { int d; };
```

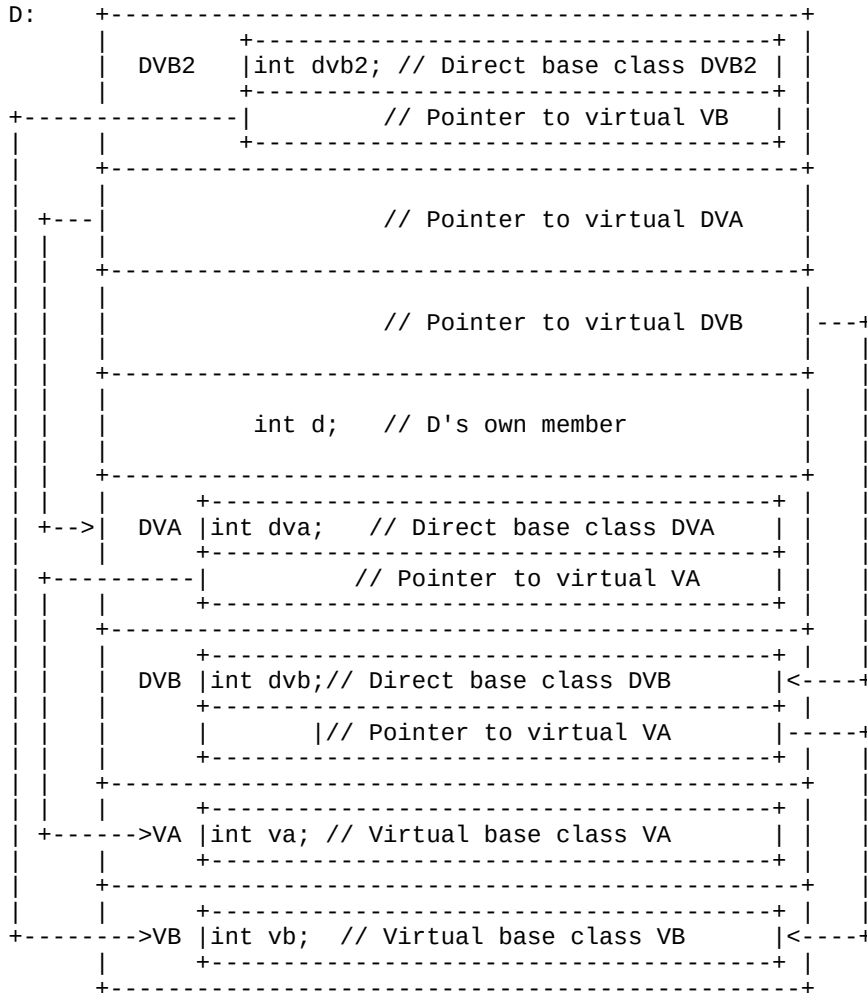
The following output shows the class lattice that the compiler produces for this example. (See toggle *Print_lattices* in Chapter 4, “Compiler Toggles” for more information.)

```

D {d}
-----
v:DVA {dva} v:DVB {dvb} v:VB {vb} DVB2 {dvb2}
-----
v:VA {va}   v:VB {vb}           v:VB {vb}
  
```


In this example, there are two pointers to virtual base VB: one within DVB and one within DVB2. When an object of class D is constructed, both are initialized to point to the single occurrence of VB; there would have been yet another pointer to VB from D, but it is optimized away. A total of five virtual-base pointers are initialized when a D object is constructed.

The following figure shows the object layout for an instance of D.

Figure 5-3. Example of Object Layout for Object Instance

Run-Time Organization

This chapter covers the following topics:

- Stack-Frame Layout
- Prologues and Epilogues
- Function Results

Stack-Frame Layout

The stack frame of a function has the format shown in the table below in the most general case:

Table 6-1. Stack Frame Format

Stack Frame		
+ (...)	Second parameter	
+8	First parameter	
+4	Caller's return address	
0	Caller's ebp	<== ebp
-4	Static Link (if nested)	
	Local variables	
	and	
	Temporaries	
	Saved registers	
	(if necessary)	<== esp

When used as such, the **ebp** register points to the base of the local stack frame and is used to address the function's parameters and its local variables and temporaries. The value at **0[ebp]** is the value of the caller's **ebp** and is used for restoring **ebp** upon exit. Just above this (in increasing memory addresses) is the return address where execution resumes after the function is finished. Above the return address are the parameters to the function.

Parameters

Each parameter to a function occupies a multiple of doublewords (four bytes) on the stack. If a parameter is of a type that is smaller than a doubleword, it is right-justified into a doubleword prior to passing; the higher bytes are not necessarily cleared. (But note that a scalar argument to a function lacking a prototype declaration is passed as type `int`, so that the upper bytes of the passed doubleword are correctly set.)

By default, parameters are pushed in reverse order, so the first parameter in the formal parameter list has the least displacement from `ebp`. (See the “Argument Passing” section in Appendix C, “Optimizing Program Performance” for more information.) In the receiving side, parameters begin at Offset + 8 from the base of the stack frame.

Static link

A function that is nested within another function is passed a static link in the `ecx` register. The static link is the address of the stack frame of the immediately enclosing function. The link is used to reference “up-level” variables, that is, locals of containing functions. If necessary, it is saved immediately below the saved `ebp`. If no static link is required, no space in the stack frame is reserved for it. Note that nested functions are an extension over Standard C, so that the static link is never encountered in Standard C programming. For more information, see Appendix F, “High C Language Extensions.”

Local variables and temporaries follow the static link, if it is present; otherwise they follow the saved `ebp`.

Saving Registers

By convention, machine registers are divided into two sets: those that are nonvolatile and those that are volatile. The former must be preserved across a call (by the callee). The latter need not be preserved.

The following registers are volatile:

eax, ecx, and edx

The following registers are nonvolatile:

ebx, ebp, esi, and edi

Direction Bit

The High C compiler assumes the direction bit is clear at all times. Functions in other languages called from High C should not leave it set.

Prologues and Epilogues

The prologue and epilogue code that the compiler generates for a function depends on whether **ebp** is used as a stack-frame base. The two conventions are shown in the following tables, and the following conventions apply:

- Lines numbered 1 allocate the stack frame.
- The line numbered 2 saves the static link and may be present only if the function is nested within another. The static link is received in the **ecx** register.
- Each line numbered 3 is necessary if the referenced nonvolatile register is directly modified within the body of the function.
- Each line numbered 4 restores a register saved in the prologue, as needed.
- Lines numbered 5 free the stack frame, if necessary, by restoring **ebp** and **esp** to their values at the point the function was entered; that is, at the first line.
- The line numbered 6 returns to the caller, who is responsible for popping the arguments from the stack.

Table 6-2. Prologue/Epilogue Instructions (if ebp contains base)

1.	pushl	%ebp	
1.	movl	%esp, %ebp	/Set up new stack frame address
2.	pushl	%ecx	/Save static link, if required
1.	subl	\$n, %esp	/Allocate locals.
3.	pushl	%esi	/Save nonvolatile register.
3.	pushl	%edi	/Save nonvolatile register.
3.	pushl	%ebx	
...			/Body goes here.
4.	popl	%ebx	
4.	popl	%edi	/Restore nonvolatile register
4.	popl	%esi	
5.	movl	%ebp, %esp	/Deallocate

5.	popl	%ebp	/locals and temporaries.
6.	ret		/Return to caller, which pops/arguments
where n is the length of the stack frame.			

The stack frame layout example listed earlier in this chapter reflects the state after the prolog code executes; that is, through step 3.

Table 6-3. Prologue/Epilogue Instructions (if ebp is not used as base)

1.	pushl	%ebp	/Save ebp if modified.
2.	pushl	%ecx	/Save static link if necessary.
1.	subl	\$n+s+c,%esp	/Allocate room for locals and register /save area and argument-list build area.
3.	movl	%esi,c(%esp)	/Save nonvolatile registers as necessary
3.	movl	%edi,c+4(%esp)	
3.	movl	%ebx,c+8(%esp)	
...			/Body goes here.
4.	movl	c(%esp),%esi	/Restore nonvolatile registers.
4.	movl	c+8(%esp),%ebx	
5.	addl	\$n+s+c,%esp	/Deallocate stack frame.
6.	ret		/Return to caller.

where:

n is the number of bytes required for locals and temporaries,

s is the size of the register save area (0, 4, 8, or 12), and

c is the length of the largest argument list built within the function.

The convention used to pass arguments depends on whether the generated code is being tuned for the 80486 (see the description of the **-486** option in the “Command-Line Options” section of Chapter 2, “Invoking the Compiler”).

If the **-386** option is specified, arguments are pushed in reverse order.

For example, for **-386** mode:


```

pushl    $3          /foo(1,2,3);
pushl    $2
    pushl    $1
    call     foo
    addl     $12,%esp  /free argument space

```

If **-486** is specified, then arguments are passed by being assigned to consecutive offsets within an argument list build area that was previously allocated at the end of the stack frame; that is, referenced by **esp**. This is the default behavior.

For example, for **-486** mode:

```
foo(1, 2, 3);
```

would be translated to:

```

    subl     $12, %esp
    .
    .
    .

    movl     $1, 0(%esp)
    movl     $2, 4(%esp)
    movl     $3, 8(%esp)
    call     foo

```

Note: These two methods of passing arguments are compatible; that is, arguments passed are in the same stack positions when `foo` gets control. The order of evaluation of the arguments is reversed. This should not affect portable code without side effects. Also note that the argument space is not deallocated after the call.

All parameters are passed by value. That means that the called function can modify its parameters without affecting values on the calling side. When a structure is passed by value, a copy of the entire structure is passed.

Function Results

The following table applies to non-structure results.

Size of Result Returned In Registers

The following are the sizes of the results returned in the specified registers:

- long long 8 bytes - *eax, edx*
- floating point results - Top of Floating Point Stack, ST0
- Four bytes— *eax*
- Two bytes— *ax*
- One byte— *al*
- All others—A temporary buffer whose address is implicitly passed by the caller as the first parameter. The callee pops this parameter

Exceptions

The following are exceptions.

When the calling convention

`_RETURN_AGGREGATE_AS_POINTER`

is used, a structured type, which would normally be returned by being placed in a temporary whose address is passed as a hidden parameter, is instead placed in a statically allocated temporary, and the address of the temporary is returned; this procedure is for compatibility with Microsoft C, Lattice C, and some UNIX C compilers. (For more information, see Chapter 8, “Externals”.)

Floating-Point Support

This chapter covers the following topics:

- The 80387 Coprocessor
- 80387 In-line Instructions

The 80387 Coprocessor

The 80387 is a fast floating-point processor that runs as a “slave” to the 80386 CPU. The 80387 has its own set of instructions that it intercepts from the host CPU.

Not every 80386 machine contains a floating-point coprocessor, but it is the responsibility of the operating system to emulate the 80387 instruction set correctly if the chip is not present. The functionality of the 80387 coprocessor chip is built into the 80486, Pentium, Pentium Pro, and Pentium II microprocessors.

80387 In-line Instructions

If toggle 387 is turned On, the compiler can generate in-line 80387 math instructions. These instructions provide the functionality in the math library (that is, *libm.a*) but with in-line hardware instructions. This improves performance, but inhibits much of the floating point exception handling that is done in the versions in the math library. Users who require this full error handling should avoid turning on this toggle. See toggle 387 in Chapter 4, “Compiler Toggles.”

The insertion of 80387 math instructions is on by default starting with SCDE 3.02.

Externals

This chapter covers the following topics:

- Interfacing to Other Languages
- Aliasing Pragmas
- Data Segmentation: the Data Pragma

Interfacing to Other Languages

The default calling convention conforms to the i386 ABI, and should be used for portability and when calling provided library functions.

To facilitate using software written in languages other than High C, the High C compiler provides a way to specify a variety of calling conventions for functions. This allows the user to link High C code with programs written in other languages, such as Pascal and FORTRAN.

The responsibility for getting data communicated properly is left to the programmer because different languages and compilers map data in different ways (see the chapters entitled “Assembly-Language Communication” and “Data Representation and Storage Mapping”), but allowing compatible calling conventions solves the most difficult problem. High C provides the **Calling_convention** pragma for that purpose.

The three forms of this pragma are as follows:

```
#pragma    Calling_convention(Expression);
#pragma    Calling_convention(Expression, _DEFAULT);
#pragma    Calling_convention();
```

The Expression must be constant and an integral type and is interpreted as the calling convention. The second form not only sets the current calling convention but also sets the default calling convention.

Note: Use upper case when spelling `_DEFAULT`.

The third form sets the calling convention to whatever it was prior to the last time it was set—that is, the first and second forms, before setting the convention, push the convention then in force on a stack, and the third form pops the stack and sets the current convention to the popped value. If the stack is empty when the third form of this pragma is encountered, as would be the case if the first use of pragma **Calling_convention** had no argument, the calling convention is undefined.

The calling convention is a bit pattern where each bit specifies a particular calling convention attribute. Two predefined names indicate the current convention and the default calling convention (to which the current calling convention is initially set):

```
_CALLING_CONVENTION    /* Current convention. */
_DEFAULT_CALLING_CONVENTION    /* Default. */
```


The calling convention has an effect on all subsequent function declarations, whether the declaration is a definition or not. The default calling convention for the High C compiler is `_REVERSE_PARMS` for **-386** compiles.

In writing the Expression, certain predefined constants can be used to obtain the bits to be combined. These constant names and their meanings are as follows:

- `NAME`
Semantics
- `_CALLEE_POPS_STACK`
The called function pops its parameters off the stack upon return. Otherwise the caller must pop the stack.
- `_RETURN_AGGREGATE_AS_POINTER`
For a function returning a structured type that would normally return its result through a pointer passed in as a hidden first parameter (see Chapter 6, “Run-Time Organization”). This attribute causes the result to be placed instead in a statically-allocated temporary, and the address of the temporary to be returned by the function as a simple pointer-sized result. This matches a typically-used UNIX structure-returning convention, but is considerably less efficient than the Standard convention used with High C.
- `_REVERSE_PARMS`
Push the parameters on the stack in reverse order. (The first parameter is pushed last.)

Undeclared Functions

C permits calling undeclared functions. The High C compiler supplies the declaration of such a function at the global level, and its calling convention is always the default calling convention.

Such undeclared functions are generally used for two purposes: to call an externally-defined function such as `printf` without bothering to define it, and to call a function declared later in the current module. If the default convention is unsatisfactory for such functions, the programmer must explicitly declare them; the explicit declarations can be encompassed by **Calling_convention** pragmas.

Duplicate Function Declarations

C permits duplicate declarations of extern functions. It is the programmer's responsibility to ensure that functions are declared with consistent calling conventions across compilation units. Within a single compilation unit, the compiler issues a diagnostic if two or more declarations of the same function have inconsistent calling conventions.

Aliasing Pragmas

Note: This is a language extension of High C, and it is non-portable.

The names of variables and functions that are communicated across module boundaries are normally made global in the resultant object module. In large programs there may be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately neither C nor most linkers provide for a structured name space; for named packages of resources, for example. Thus the well-chosen “internal” names in a program may not also be usable as “external” names (those known to the linker) as they should be. Some method of aliasing internal names to externals is needed, and High C provides it.

It is important to be able to alias such names to avoid conflicts in the linker’s external symbol dictionary, rather than being forced to pervert the internal names themselves. The internal names must be well-chosen “containers of meaning,” for program maintainability.

Alias

This pragma specifies, for a given internal name, another name for external or public purposes. It is the alternate name that appears in the object module. The form of the **Alias** pragma is as follows:

```
pragma    Alias(<Internal_name>,<External_name>);
```

where <Internal_name> is the function or variable identifier being aliased and <External_name> is a constant string expression whose value denotes the alternate or external name.

The **Alias** pragma must appear in the scope of the declaration of the internal name.

Example

```
void Initialize();  
pragma Alias(Initialize, "x_initialize");  
    /* The function Initialize is referenced in the */  
    /* object-module symbol table as "x_initialize". */  
int BA;  
pragma Alias(BA, "Ph.D.");  
    /* "BA" Referenced in the symbol table as "Ph.D.". */
```

<External_name> need not be a legal identifier, thus providing a way to define virtually any symbol (any symbol acceptable to the linker).

Data Segmentation: the Data Pragma

Note: Read this section if you are interested in using a data communication convention different from that of Standard C. This is a non-portable language extension of High C.

Communication between separately compiled modules is achieved by using the extern storage class in C. Multiple defining declarations of a variable *x* are allowed, as long as, at most, one of them initializes *x* (in which case, the extern storage class is not required).

The **Data** pragma provides an alternative method of sharing data, using named blocks. Its general usage is illustrated by:

```
pragma    Data(class, "blockname");
int       X, Y, Z;
...       /* Other normal C declarations may */
          /* appear here. */
pragma    Data;
          /* "Turns off" the prior Data pragma. */
```

where *class* is one of Common, Import, or Export, and “*blockname*” is a constant string expression. The ending **Data** pragma has no parameters.

Only the given block name is made known to the linker as a global symbol: each variable is addressed at a fixed offset within the block.

When the Import class is specified, the symbol is given the undefined global attributes and a value of zero; when Export, the symbol is defined in the module’s bss or data segment and given the global attribute.

When Common, the symbol is flagged as a named common block, that is, given the undefined global attributes and a value that is equal to its length.

Scope

Each **Data** pragma must be terminated or “turned off” as illustrated above in the same scope in which it is turned on. The storage class specification applies only to variable declarations between the specification and its termination, but not to any variables declared within enclosed function definitions. That is, variables declared within an enclosed function definition are not affected; at a function definition, any active **Data** pragma temporarily becomes inactive and the default applies through the end of the function.

A compile-time warning is issued if a **Data** pragma is specified when a prior **Data** pragma is still active (in which case the subsequent pragma applies), or if a **Data** pragma is active at the end of a function declaration or at the end of a compilation unit. Thus **Data** pragmas cannot be nested within a single function, though they can be nested if they apply to the local variables of distinct functions.

Example

```
pragma      Data(Common, "block");
int         Tables_are_loaded;
    struct   {...} Tables;
    pragma   Data;
```

The names *Tables* and *Tables_are_loaded* are mapped at consecutive displacements (subject to boundary alignment) within the common block “block.”

Assembly Language Communication

The following topics are included in this chapter:

- Assembly Routines
- Function Naming Conventions
- Naming Conventions in C++
- Calling Assembly Routines from High C/C++
- Examples: Calling Assembly Routines from C or C++
- Calling C or C++ from Assembly
- Data Communication
- In-line Assembly Code

Assembly Routines

The “Prologue and Epilogue” section in Chapter 6, “Run-Time Organization” describes the code that an assembly routine must execute in order to be callable from C. In short, an assembly routine should be coded according to the following guidelines. Substitute appropriate values for symbols in *italics*.

```
.text
.globl    name
name:
    push    %ebp
    mov     %esp, %ebp
    sub     $n, %esp
    push    %esi    [As needed]
    push    %edi    [As needed]
    push    %ebx    [As needed]

/ The body of the routine goes here.

    pop     %ebx    [As needed]
    pop     %edi    [As needed]
    pop     %esi    [As needed]
    mov     %ebp, %esp
    pop     %ebp
    ret      /Return to caller,
              /who pops arguments.
```

where *name* is the function’s name as referenced from C; and *n* is the size of the local stack frame.

esi, **edi**, and **ebx** are pushed and popped in order to preserve register variables. This is necessary only if the caller expects the assembly-language routine to do so. By default, High C functions expect nonvolatile registers to be saved. They are **esi**, **edi**, **ebx**, and **ebp**.

The High C compiler assumes the direction bit is clear at all times. Therefore the assembly routine should not leave it set.

To see how arguments are passed and how function results are returned, see Chapter 6, “Run-Time Organization.”

Function Naming Conventions

A function that is global (that is, callable across module boundaries) must have information provided to the linker that associates its name with its address. This is done by placing a corresponding name in the symbol table of the object module and giving it the global attribute. This name need not be the same as the actual name of the function; it need only be derivable from it. The symbol table entry is called the external presentation of the internal name.

In both assembly language and C, the external presentation is lexically identical to the internal presentation.

The **Alias** pragma can be used to explicitly specify an alternate external presentation.

High C/C++ provides two pragmas for changing external presentations of internal names:

- You can use pragma **Global_aliasing_convention** to cause all external names to be altered: to be prefixed by an underscore, for example, or to be uppercased.
- You can use pragma **Alias** to define an alternate external presentation of a single identifier (that is, to explicitly define how a public symbol is to appear in the symbol table).

The next two code fragments illustrate two ways to call an assembly function names `_fnc` from a High C/C++ program.

Using pragma **Alias**:

```
extern fnc();
...
/* Call '_fnc' instead of default 'fnc'. */
#pragma Alias (fnc, "_fnc")
void main () {
    ...
    fnc();
    ...
}
```

Using pragma **Global_aliasing_convention**:

```
/* Add '_' prefix to all symbol-table names. */
#pragma Global_aliasing_convention ()
...
```

```
void main () {  
    ...  
    fnc (); /* Call '_fnc' */  
    ...  
}
```

Naming Conventions in C++

In C++, function names in the symbol table are in mangled form. The compiler derives the mangled name from the function's source-code name, its argument types, and any enclosing class definitions. It is difficult to access a C++ function from assembly language because of the name mangling.

To access an assembly-language function from C++, declare the function as a C function using `extern "C"`. The compiler does not mangle C function names.

Calling Assembly Routines from High C/C++

Example 2 in the following section demonstrates how to make a call to assembly language function `peek()` from High C/C++ code.

Examples: Calling Assembly Routines from C or C++

Example 1

High C:

```
extern
void intersect(char *dest, char *src, int len);
void main() {
    char a[256], b[256]; ...
    intersect(a, b, 256); ...
}
```

Assembly:

```
.text
.globl    intersect
intersect:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %esi
    pushl    %edi
    movl     8(%ebp), %esi
    movl     12(%ebp), %edi
    movl     16(%ebp), %ecx
    jcxz     nothing_to_do
L:    movb    (%edi), %al
    andb     (%esi), %al
    stob     / Intel mnemonic: stosb
    incl     %esi
    loop     L
nothing_to_do:
    popl     %edi
    popl     %esi
/ Could use leave for these two instrs on 286/386:
    movl     %ebp, %esp
    popl     %ebp
    ret
```

Example 2

This example illustrates how the saving and restoring of the **ebp** register can be avoided for simple routines.

High C:

Examples: Calling Assembly Routines from C or C++

```
extern char peek(char *adr);
void main(){
    char b; ...
    b = peek((char *)0x8000);
    ...
}
Assembly:
    .text
    .globl    peek
peek:
    movl     4(%esp),%eax    /Get arg #1
    movb     (%eax),%al      /set up return value
    ret                          /and return
```

Calling C or C++ from Assembly

To call a function from assembly language, first push the arguments on the stack in right-to-left order and then execute a call instruction. The arguments are then popped from the stack by incrementing **esp** by the appropriate amount.

Example

High C:

```
void      write_string(char *s) {
    printf("%s\n",s);
}
```

Assembly:

```
.text
.globl    write_string
...
push     $msg      / Intel: offset msg.
call     write_string
pop      %eax      / Remove arg.
...
.data
msg:
.string   "This is a message."
```

Data Communication

The external presentation of a global variable's name in High C is determined the same way and can be overridden in the same way as for external functions.

Chapter 5, "Data Representation and Storage Mapping," explains how the various C data types are mapped into storage. Note that uninitialized global variables without the `extern` qualifier are actually defined as individual common blocks. The following examples illustrate the sharing of variables between C and assembly:

High C:

```
int alpha,beta;
char hextable[] = "0123456789abcdef";
extern short status;
Assembly:
    .data
        .comm    alpha,4,4
        .comm    beta,4,4
        .align   4
        .globl   hextable
hextable:
        .string  "0123456789abcdef";
        .globl   status
        .text
/ Usage:
    mov     alpha,%eax
    add     beta,%eax
```

High C provides the ability to map more than one variable into a named block, for instance, a common block as in FORTRAN. This facility is provided by the **Data** pragma and is documented in the chapter entitled "Externals." The following example illustrates how such a common block can be accessed from assembly language.

C Common Block Definition:

```
pragma    Data(Common, "BLOCK");
int       a,b;
char      c,d;
short     e;
pragma    Data;
Assembly:
        .comm    BLOCK,12,4
        .set     a,0
```

```

        .set      b,4
        .set      c,8
        .set      d,9
        .set      e,10
        ...
/ Usage:
    movl    BLOCK+a,%eax / Load value of a.
    movl    BLOCK+b,%edx / Load value of b.
    movb    BLOCK+c,%cl  / Load value of c, etc.

```

Note that variables a,b,c,d, and e are not global; that is, they have no external presentation. The only name that has an external presentation is BLOCK.

In-line Assembly Code

A simple in-line assembler directive is provided. A function call of the form `asm("string")` places the string "string" in the corresponding place in the generated assembly file. A newline character is appended. No attempt is made to check the contents of the string. This directive may appear only where a function call is permitted; that is, within a function definition.

Note: The string must not alter control flow (that is, contain jumps), nor modify any machine register that is available for general use by the compiler, or reference fixed offsets in the stack frame.

See also "Enhanced ASM Facility," in Appendix E, "Other Constructs and Utilities" for a more flexible and usable way to directly insert assembler code constructs into a program.

Listings

This chapter on Listings contains the following topics:

- Pragmas Page, Skip, Title
- Format of Listings
- Example

Pragmas Page, Skip, Title

To cause **n** page ejects at some point in the listing, insert:

```
pragma Page(n); /* n is the number of ejects.*/  
To cause n lines to be blank at some point in the listing,  
insert:  
pragma Skip(n); /* n is the number of blanks.*/  
To cause a title T to appear at the top of each successive  
page, place the following pragma in the source:  
pragma Title(T); /* T is a string constant.*/
```

Each successive **Title** pragma changes the title for subsequent pages. The only way to title the first page is to place the **Title** pragma in the profile and use pragma **On(List)**; at the end of the profile, or wherever the listing should start in the source file. When the **-Hlist** option is specified on the command line, the first page is untitled, because the listing starts before the **Title** pragma is seen by the compiler. See Chapter 3, “Compiler Pragmas.”

Format of Listings

Ruler

The first line after any header and title lines on each page is a “ruler” that defines three fields for each line. The fields are:

- Three level numbers
- The line number
- The line contents

The ruler appears as follows:

```
Levels
LINE#| ---+---1---+---2---+---3---+---4---+---5---+---6---...
```

Level numbers can be used to find a missing } or comment terminator when a message such as “Unexpected end-of-file” is produced by the compiler. All three level numbers are initially zero, but they are printed as blanks rather than 0.

The first level number indicates the scope nesting level for struct or union declarations. The second level number indicates the statement nesting level. It is incremented at the beginning of each { and is decremented at the corresponding }.

The third level number indicates the structure initialization nesting level. It is incremented at the beginning of each { and decremented at the corresponding }.

Include files

A first level Include file named File_name is indicated as starting after a line containing “+(File_name” in the line-number field, and ending just before a matching “+)File_name” line. The included lines have “+” in the leftmost column of the line-number field, and those lines are numbered independently of the main source file. Include file

An Included file inside an Include file has an extra “+” on each of its lines for each level of inclusion, except that line numbers take precedence over “+”s in the line-number field if the “+”s would otherwise intrude into the field.

The profile, if any, is listed as an Include file when a listing has been requested.

The listing facility should be used in conjunction with the **-Hnocpp** option. Otherwise, the output of the outboard C preprocessor will be listed. Each Include file specified with the #include preprocessor statement is back-substituted with no indication on the listing.

Example

The following sample program listing is enhanced with boldface reserved words and followed by a sample assembly deck.

```
cc -S -Hanno -O -Hlist queens.c MetaWare High C Compiler 2.3 01-
Jan90 03:24:42 queens.c Page 1
Copyright (C) 1983-90 MetaWare Incorporated.
Target: UNIX/386
Levels LINE #|-----1-----2-----3-----4-----5-----+
1 | /* From Wirth's Algorithms+Data Structures=Programs. */
2 | /* This program is suitable for code-generation */
3 | /* benchmark, especially given common sub-expressions */
4 | /* in array indexing. See the Programmer's Guide for */
5 | /* how to get a machine code interlisting. */
6 |
7 | pragma Title("Eight Queens problem.");
8 |
9 | typedef enum {False,True} Boolean;
10 | typedef int Integer;
11 |
12 | #define Asub(I) A[(I)-1] /* C's restriction that array */
13 | #define Bsub(I) B[(I)-2] /* indices start at zero */
14 | #define Csub(I) C[(I)+7] /* prompts definition of */
15 | #define Xsub(I) X[(I)-1] /* macros to do subscripting. */
16 | /* Pascal equivalents: */
17 | static Boolean A[ 8]; /* A:array[ 1.. 8] of Boolean */
18 | static Boolean B[15]; /* B:array[ 2..16] of Boolean */
19 | static Boolean C[15]; /* C:array[-7.. 7] of Boolean */
20 | static Integer X[ 8]; /* X:array[ 1.. 8] of Integer */
21 |
22 | void Try(Integer I, Boolean *Q) {1 1
23 |   Integer J = 0; 1 1
24 |   do {2 2
25 |     J++; *Q = False; 2 2
26 |     if (Asub(J) && Bsub(I+J) && Csub(I-)){ 3 3
27 |       Xsub(I) = J; 3 3
28 |       Asub(J) = False; 3 3
29 |       Bsub(I+J) = False; 3 3
30 |       Csub(I-J) = False; 3 3
31 |       if (I < 8) { 4 4
32 |         Try(I+1,Q); 4 4
33 |       if (!*Q) { 5 5
34 |         Asub(J) = True; 5 5
35 |         Bsub(I+J) = True; 5 5
36 |         Csub(I-J) = True; 5 5
37 |       } 4 4
```

Example

```

38 |         } 3 3
39 |                                     else *Q = True; 3
40 |                                     } 2 2
41 |                                     } 1 1
42 |         while (!(*Q || J==8)); 1 1
43 |     }
44 | #pragma Page(1); /* Page eject requested.*/

MetaWare High C Compiler 2.3 01-Jan-90 03:24:42 queens.c Page2
      Eight Queens problem.
Levels  LINE  # |-----1-----2-----3-----4-----5---
--+
45 | void main () {1 1
46 |     Integer I; Boolean Q;1 1
47 |     printf("%s\n", "go");1 1
48 |     for (I = 1; I <= 8; Asub(I++) = True);1 1
49 |     for (I = 2; I <= 16; Bsub(I++) = True);1 1
50 |     for (I = -7; I <= 7; Csub(I++) = True);1 1
51 |     Try(1,&Q);1 1
52 | #pragma Skip(3); /* Skip 3 lines. */1 1
53 |     if (Q)1 1
54 |         for (I = 1; I <= 8;) {2 2
55 |             printf("%4d", Xsub(I++));2 2
56 |         }1 1
57 |     printf("\n");1 1
58 | }

```

The assembly file is named *queens.s*. Its contents are as follows:

```

.file      "queens.c"
.optim
.globl     .oV0
.set       .oV0,0
.globl     .oVhc2.2a
.set       .oVhc2.2a,0
.lcomm     A,8
.lcomm     B,15
.lcomm     .L00990003,1
.lcomm     C,15
.lcomm     .L00990004,1
.lcomm     X,32
.text
.align     4
.align     4
/===== Try =====/
.globl

```

```

Try
Try:
/1 : /* From Wirth's Algorithms+Data Structures = Programs. */
/2 : /* This program is suitable for a code-generation */
/3 : /* benchmark, especially given common sub-expressions */
/4 : /* in array indexing. See the Programmer's Guide for */
/5 : /* how to get a machine code interlisting. */
/6 :
/7 : pragma Title("Eight Queens problem.");
/8 :
/9 : typedef enum{False,True} Boolean;
/10 : typedef int Integer;
/11 :
/12 : #define Asub(I) A[(I)-1] /* C's restriction that array*/
/13 : #define Bsub(I) B[(I)-2] /* indices start at zero */
/14 : #define Csub(I) C[(I)+7] /* prompts definition of */
/15 : #define Xsub(I) X[(I)-1] /* macros to do subscripting.*/
/16 : /* Pascal equivalents: */
/17 : static Boolean A[ 8]; /* A:array[ 1.. 8] of Boolean */
/18 : static Boolean B[15]; /* B:array[ 2..16] of Boolean */
/19 : static Boolean C[15]; /* C:array[-7.. 7] of Boolean */
/20 : static Integer X[ 8]; /* X:array[ 1.. 8] of Integer */
/21 :

/22 : void Try(Integer I, Boolean *Q) {
        pushl    %ebp
        movl     %esp,%ebp
        subl     $4,%esp
        pushl    %ebx
        pushl    %esi
        pushl    %edi
/23 :     Integer J = 0;
            subl     %esi,%esi
            movl     12(%ebp),%ebx
            .L001002:
/24 :     do    {
/25 :             J++; *Q = False;
            incl     %esi
            movb     $0x0,(%ebx)
            subl     %eax,%eax
/26 :     if (Asub(J) && Bsub(I+J) && Csub(I-J)) {
            cmpb     $0x0,A-1(%esi)
            je       .L001009
            movl     8(%ebp),%edx
            leal     (%edx,%esi),%edi
            cmpb     $0x0,B-2(%edi)
            je       .L001009
            movl     %edx,%ecx
            subl     %esi,%ecx
            movl     %ecx,-4(%ebp)
            cmpb     $0x0,C+7(%ecx)

```

Example

```

je .L001009
/27 : Xsub(I) = J;
      movl    %esi,X-4(,%edx,4)
/28 : Asub(J) = False;
      movb    $0x0,A-1(%esi)
/29 : Bsub(I+J) = False;
      movb    $0x0,B-2(%edi)
/30 : Csub(I-J) = False;
      movl    -4(%ebp),%ecx
      movb    $0x0,C+7(%ecx)
/31 : if (I < 8) {
      cmpl    $0x8,%edx
      jge     .L001008
/32 :     Try(I+1,Q);
      pushl   %ebx
      incl    %edx
      pushl   %edx
      call    Try
/33 : if (!*Q) {
      movb    (%ebx),%al
      addl    $8,%esp
      andb    %al,%al
      jne     .L001009
/34 :     Asub(J) = True;
      movb    $0x1,A-1(%esi)
/35 :     Bsub(I+J) = True;
      movb    $0x1,B-2(%edi)
/36 :     Csub(I-J) = True;
      movl    -4(%ebp),%ecx
      movb    $0x1,C+7(%ecx)
      jmp     .L001009
      .align  4
      .L001008:
/37 :     }
/38 : }
/39 : else *Q = True;
      movb    $0x1,(%ebx)
      movb    $0x1,%al
      .L001009:

/40 : }
/41 : }
/42 : while (!(*Q || J==8));
      andb    %al,%al
      jne     .L001011
      cmpl    $0x8,%esi
      jne     .L001002
      .L001011:
/43 : }
      popl    %edi
      popl    %esi
      popl    %ebx

```



```

        leave
        ret
        .data
        .align      4
        .align      4
.L1: .string      "%s\n"
        .align      4
.L2: .string      "go"
        .align      4
.L3: .string      "%4d"
        .align      4
.L4: .string      "\n"
        .text
        .align      4
/-----| main |-----/
        .globl      main
main:
/44 :      pragma Page(1); /* Page eject requested. */
/45 :      void main () {
        pushl      %ebp
        movl      %esp,%ebp
        subl      $4,%esp
        pushl      %ebx
        pushl      %esi
/46 :      Integer I; Boolean Q;
/47 :      printf("%s\n","go");
        pushl      $.L2
        pushl      $.L1
        call      printf
/48 :      for (I = 1; I <= 8; Asub(I++) = True);
        movl      $0x1,%eax
        addl      $8,%esp
        .L002002:
        movb      $0x1,A-1(%eax)
        incl      %eax
        cmpl      $0x8,%eax
        jle      .L002002
/49 :      for (I = 2; I <= 16; Bsub(I++) = True);
        movl      $0x2,%eax
        .L002004:
        movb      $0x1,B-2(%eax)
        incl      %eax
        cmpl      $0x10,%eax
        jle      .L002004
/50 :      for (I = -7; I <= 7; Csub(I++) = True);
        movl      $0xffffffff9,%eax
        .L002006:
        movb      $0x1,C+7(%eax)
        incl      %eax
        cmpl      $0x7,%eax
        jle      .L002006

```

Example

```
/51 :    Try(1,&Q);
        leal        -1(%ebp),%eax
        pushl       %eax
        pushl       $0x1
        call        Try
/52 :    pragma Skip(3); /* Skip 3 lines. */
/53 :    if (Q)
        addl        $8,%esp
        cmpb        $0x0,-1(%ebp)
        je          .L002010
/54 :    for (I = 1; I <= 8;) {
        movl        $0x1,%esi
        leal        .L3,%ebx
        .L002009:
/55 :        printf("%4d",Xsub(I++));
        pushl       X-4(,%esi,4)
        incl        %esi
        pushl       %ebx
        call        printf
        addl        $8,%esp
        cmpl        $0x8,%esi
        jle         .L002009
        .L002010:
/56 :    }
/57 :    printf("\n");
        pushl       $.L4
        call        printf
/58 :    }
        addl        $4,%esp
        popl        %esi
        pop         %ebx
leave
ret
```

Compiling and Linking C++

This chapter presents information specific to compiling and linking C++ programs. It contains the following sections:

- Compiling and Linking C and C++ Modules
- Using the Built-In Inliner Versus the Stand-Alone Inliner
- Standard Template Library (STL)
- Generating Run-Time Type Information (RTTI)
- Switching to New-Style Type Cast Notation
- Limitations in Pointer-to-Member Comparison
- Using Templates
- Instantiating Templates Manually

Compiling and Linking C and C++ Modules

The High Performance C/C++ compiler includes both a C compiler and a C++ compiler in one executable. C++ has its own set of standard header files that are distinct from those of C. C++ also has its own run-time library. The compiler will automatically include the C++ header files when compiling a C++ source module, and by default will add the C++ link library during linking.

The C++ support library contains several class libraries associated with traditional C++ implementations, including:

- I/O streams classes, including **ios**, **streambuf**, **istream**, **ostream**, **stringstream**, **fstream**, and **iostream**
- Complex number class
- New dynamic allocation class
- Typeinfo class
- Exception handling support
- Standard template library (STL)

The header files for the library exist in */usr/ccs/incc*, and the library exists in the */usr/ccs/lib* directory as **libcc.a**.

The driver checks the file extension of the source file you specify on the command line to determine whether it should be compiled as C or C++ code. By default, any source files that end in **.C**, **.cpp**, **.cc**, or **.ii** will be compiled as C++ programs. Other file naming extensions can be recognized by using the **-Hcpp=ext** option, for example **-Hcppext=c** would recognize 'x.c' as a C++ source file and would compile it in C++ mode.

Do not use these identifiers in C

The C++ library files contain identifiers that intrude upon the name space of ANSI Standard C programs. Identifiers with these names should not be used in C if you are going to mix C with C++:

- **cerr**
- **cin**

- **clog**
- **cout**

Symbols

`__CPLUSPLUS__` and `__cplusplus`

Symbolic constants `__CPLUSPLUS__` and `__cplusplus` are defined when you compile C++ modules. You can use these constants in your programs to conditionally compile C or C++ code.

External Linkage between C and C++

When C++ symbol names are created they have additional information appended that indicates the type of the object. This makes it possible to do type safe linking, as the type embedded in the symbol name must match the type of the object to resolve the symbol reference. This makes calls to functions defined in other modules safer, as the expected number and type of each argument must match, and enables several of the C++ language features, like function overloading.

C does not do this, so references to C objects must not have the type information appended or link failures will occur. Programs written in C++ that reference C symbols must define these symbols to the compiler specially as C . symbols.

C++ has a way to define these symbols using the ‘extern “C” { ... }’ definition. The definitions between the brackets are defined to use C external linkage conventions which omit the object type information postfixes. Any C library references or other library or object references to programs compiled in C must use this convention.

Automatic “wrapping” of C include file definitions

By default, any files included from the standard C header files directory */usr/include* and all directories below it are “wrapped” with ‘extern “C” {’ at the top of the include file and ‘}’ at the bottom. This is done so that references to the C library or other C support libraries are done with C type external linkage.

The **Hc_wrap=directory** option permits additional include file directories to be wrapped for C external linkage. See Chapter 2, “Command Line Options.”

Anytime a directory containing header files that contain C external functions and data variable declarations is included in a C++ compilation, the directory must be wrapped. For example, a directory with header files for a library of C functions will need to be wrapped if a C++ compile includes function calls to it.

Using the Built-In Inliner Versus the Stand-Alone Inliner

The High C/C++ compiler has an optional optimization phase known as the function inliner, or inliner for short. This stand-alone inliner is described in Appendix D, “Inliner Function.”

The Optional Stand-Alone Function Inliner

You can use the stand-alone inliner to inline a function anywhere, even if the function is called prior to its own definition (normally, such forward referencing is an error in C++ programs).

This inlining is especially helpful with functions in template classes, because the expansion of separately defined class template functions must be delayed until the end of compilation, in case you have overridden a template function with a specific version of your own. Thus, calls to the template function are emitted by the compiler before the function definition is seen.

When to use the stand-alone inliner

In most cases, you will not need to use the stand-alone inliner for C++ functions. Use the stand-alone inliner only if you want to inline the following:

- arbitrary functions that are not necessarily designated as inline
- functions whose bodies are presented after the compiler sees calls to those functions
- functions across compilation units (via "wide inlining")

See Appendix D, “Inliner Function” for more information.

The Built-in C++ Inliner

The High C/C++ compiler also includes a built-in inliner that provides most of the functionality of the stand-alone inliner. The inlining of C++ inline functions is performed automatically, so you do not need to invoke the stand-alone inliner to inline such functions.

Size limit for inlining C++ functions

The compiler limits the size of functions that are automatically inlined, to limit code size expansion in compiled code. You can change the maximum size of functions to inline with command-line option **-Hinlsize**. See Chapter 2, “Invoking the Compiler.”

If you want a class function to be inlined but plan to present the body later, specify inline in both the class declaration and in the function definition.

```
struct s {
    inline int fnc();
};
main () {
    s x;
    x.fnc();
}
inline s::fnc() { return 0; }
```

Using the Name Unmanglers

The compiler converts the name of each function to a *mangled name* that encodes the function type in the text of the name. This encoding is necessary in C++ to support overloading multiple functions with the same name.

Type-Safe Linkage

Name mangling makes type-safe linkage possible. This means you can safely call a function that is defined in another module, and not worry whether that function is declared with the same parameter types in the other module.

Converting Mangled Names

The name unmanglers *unmangle*, *demangle*, and *c++filt*, convert the mangled function and variable names generated by the C++ compiler into a form similar to the C++ declaration syntax. You can use them as filters to read linker output or assembler output. You can also use them to help you perform other essential debugging-related tasks.

Each unmangler deciphers names generated by the High C++ compiler, looking for certain key sequences in a name to determine whether it is a mangled name.

For example, names containing embedded `__of` or `@` strings are treated as mangled names.

Invoking the Name Unmangler

You can invoke a name unmangler from the command line either with or without arguments:

- **Method 1** Invoking the name unmangler with arguments.
The command-line arguments are unmangled. You can supply any number of arguments; the unmangled form of each argument is printed to `stdout`, each on a separate line.
- **Method 2** Invoking the name unmangler without arguments.
If there are no command-line arguments, the unmangler specified reads data from `stdin`, parses the data into words (delimited by whitespace — spaces, tabs, or newlines), and attempts to unmangle each word. This technique allows the unmangler to be used as a filter.

Examples of Name Unmangling

Given these C++ function declarations:

```
int fnc( int i1, int i2 );
void fnc( short i1, short i2 );
```

the compiler mangles the function names like this:

```
__fnc__FiT1
__fnc__FsT1
```

and `unmangle` unmangles them like this:

```
fnc(int,int)
fnc(short,short)
```

The unmangled names are fully qualified with class membership scope, if applicable. For example, this command:

```
unmangle _j__1XFi
```

writes this to `stdout`:

```
x::j(int)
```

Note: `unmangle` does not try to convert names that do not match the format of a mangled name. Such names are simply echoed without modification.

CAUTION: `unmangle` expects mangled names to be correctly constructed. If there is an error in the mangled name, `unmangle` might get confused and produce unexpected output. This feature makes command-line usage hazardous, because a typographical error on the command line could result in an incorrect unmangled name.

Using Exception Handling

This section describes how to compile classes that are exception-aware, and how to use them with existing classes that are not exception aware.

Generating Exception-Aware Classes

An important part of exception handling is the clean-up of partially constructed or destructed objects. To properly achieve the clean-up, constructors increment a global “subobject count” by one and destructors decrement the count by one. For example, after construction of an object with a total of three subobjects, the subobject count is incremented by three.

A class is exception-aware if it’s constructors increment the global subobject count and the destructor decrements the count.

Making a Class Be Exception-Aware

By default, a class is not exception-aware. You make it so by turning On toggle *Exception_aware_class*. Exception-aware classes can be intermixed with non-exception-aware classes within a class hierarchy.

Completely Constructed Objects

Exception Unaware

For a completely constructed object, no portion of which is exception-aware, no destructors are run during clean-up. This is an advantage, because it saves time during clean-up.

Somewhat Exception Aware

For a completely constructed object, a portion of which is exception-unaware, all destructors are run during clean-up.

Partially Constructed Objects

For partially constructed objects, the presence of non-exception-aware classes introduces imprecision in the clean-up.

When a Partially Constructed Object is Fully Constructed

A partially constructed object is considered to be fully constructed if all its exception-aware subobjects have been constructed. All destructors are run, even though the object is partially constructed.

For example, if all but the most-derived constructor completes, and the most-derived class is not exception-aware, the object appears fully constructed.

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
struct B { ~B(); };
#pragma Pop(Exception_aware_class)
struct C:A, B { ~C(); C() { throw 1; } };
C x;
```

Even though C's constructor throws the exception, **x** is taken as fully constructed and the C subobject is fully destructed.

Ensuring a Fully Constructed Object

If you are concerned about a destructor running on a non-exception-aware class, be sure that the most-derived class is exception-aware. In that way, you ensure that if all the exception-aware objects are constructed, the object is fully constructed.

For a partially constructed object with some exception-aware subobjects not constructed, a non-exception-aware subobject is not destructed unless its construction was followed by the successful construction of an exception-aware subobject.

For example:

```
#pragma On(Exception_aware_class)
struct A { ~A(); };
#pragma Pop(Exception_aware_class)
struct B { ~B(); };
struct C { A a1; B b; A a2; };
C x;
```

If the construction of the second A-type class, a2, throws an exception, the b subobject is not destructed, even though it was fully constructed.

When to Make a Class be Exception-aware

You should make a class be exception-aware if either of the following is true:

- You do not want the destructor to run on an unconstructed object; for example, if the destructor accesses a pointer allocated during construction.
- You want the destructor always to run on a constructed object; for example, if the destructor frees storage allocated during construction.

Interacting with Previously Compiled Classes

Option -Hexcept

Option **-Hexcept** makes it safer to use toggle *Exception_aware_class*. Specifying **-Hexcept** turns On toggle *Exception_aware_class*. The runtime library is exception-aware by default.

Wrapping Header Files

Header files for third-party libraries that do not contain exception-aware classes must not be compiled with toggle *Exception_aware_class* turned On (whether or not you use option **-Hexcept**).

You must protect these header files by wrapping them with other include files that turn Off and On toggle *Exception_aware_class*. To see how this works, specify **-Hon=list** when you compile.

Option: **-Hnonexcept_wrap**

To direct the compiler to wrap an include file, insert a character string **xxxx** in the file's name, then specify option **-Hnonexcept_wrap=xxxx** when you compile. For example, given this command line:

```
cc fnc.cpp -Hexcept -Hnonexcept_wrap=nonexc
```

any include file with the string "nonexc" in its name is wrapped with *#pragma Off(Exception_aware_class)* at the beginning and *#pragma Pop(Exception_aware_class)* at the end.

Option **-Hnonexcept_wrap** behaves just like option **-Hc_wrap**, except that option **-Hnonexcept_wrap** uses wrapper files called *exc_off.h* and *exc_pop.h*.

Standard Template Library (STL)

The Standard Template Library (STL) is a general purpose library of C++ foundation class structures, and algorithms to manipulate the structures easily and effectively. It is built upon C++ templates and generic template manipulation algorithms.

The STL is available as public domain software, and is being considered as the foundation for the ANSI C++ libraries. The components are general purpose building blocks for creating complex applications. To work in our environment, the version of STL in Version 3.03 of the High C Compiler contains minor modifications to the original STL versions, and is fully supported.

For more information on the STL and its use, see the publication *STL Tutorial and Reference Guide* by Musser and Saini (Addison Wesley).

Generating Run-Time Type Information (RTTI)

Non-class Types

For a non-class type, use `typeid` to generate RTTI.

Class Types

For a class type, turn On toggle *Run_time_type_info* to ensure that RTTI is generated and stored in the class's virtual-function tables. (The default for toggle *Run_time_type_info* is On).

For such a class D, polymorphic `typeid`s on a subobject of D return the type object for D, and `dynamic_cast` works for such subobjects. Also, classes satisfying the single-copy heuristic have just one copy of the RTTI. For a discussion of the single-copy problem, see the section entitled “The Single-Copy Problem.”

When toggle *Run_time_type_info* is Off, RTTI is generated only as needed by `typeid`, so you cannot guarantee the presence of a pointer to the RTTI in the virtual-function tables. Thus, a polymorphic `typeid` can return a NULL reference.

In particular, a compilation unit that generates virtual-function tables for the class, but lacks a reference to `typeid`, also lacks a pointer to the RTTI in the virtual-function table.

When Not to Use Toggle *Run_time_type_info*

It is best to leave *Run_time_type_info* turned On for all your classes. Be aware, however, of two things:

- You should not turn On toggle *Run_time_type_info* globally for header files from third parties.

Because RTTI is a new feature, third-party libraries might not contain RTTI. If you turn the toggle On around such headers, the compiler might generate a reference to the RTTI for a class satisfying the single-copy heuristic when it generates the RTTI for one of your classes, and your program link will fail.

- You might want to turn Off toggle *Run_time_type_info* around classes with virtual functions that do not satisfy the single-copy heuristic.

For such classes, the RTTI and virtual-function tables are generated for every compilation unit containing the definition of the class, even if the class is not used. For these classes, the compiler warns that each compilation unit will have a copy of the virtual-function tables.

Switching to New-Style Type Cast Notation

The proposal of the *X3J16/96-0018 ANSI C++ Draft Standard* for the new-style cast-notation syntax suggests the following:

When moving from the old-style casts to the new, replace all old-style casts with the new notation **static_cast**, and see what diagnostics you get when you recompile.

To facilitate this replacement process, High C/C++ provides the toggle *Try_static_cast* (default Off). The effect of this toggle is to treat all C++ casts of the form **(Type)expr** as **static_cast**, and warn at each line containing a cast (so you can find them all). An error might also be generated if the **static_cast** is not valid.

Limitations in Pointer-to-Member Comparison

When a pointed-to member function is a virtual function, a comparison between two pointer-to-member functions might not yield the results you expect.

If one source file sets the value and a different source file does the comparison, the comparison fails because different function "thunks" are used to implement the call to the virtual function.

File *hdr.h*:

```
struct s {  
    virtual fnc();  
};
```

File *1.cc*:

```
#include "hdr.h"  
extern void (s::*p)();  
extern sub();  
main () {  
    sub(); // Set the pointer  
    if (p==s::fnc)  
        printf("OK\n"); // Will not print OK  
}
```

File *2.cc*:

```
#include "hdr.h"  
void (s::*p)();  
sub() {  
    p = s::fnc;  
}
```

Using Templates

Templates are a powerful feature of C++. With templates you can define generic functions or classes. Unlike macros, which are sometimes used to do the same thing (for example, *generic.h*), instantiations of templates are accompanied by full type checking.

The Annotated C++ Reference Manual makes reference to "function templates" and "template functions."

Function templates and template functions:

A template function is an instantiation of a function template, so we use the term "instantiation" or "instance" rather than "template function."

Class templates and template classes:

The terms "class templates" and "template classes" are related the same way. A template class is an instantiation of a class template, so we use the term "instantiation" or "instance" rather than "template class."

The rest of this section explains how to use templates, discusses different types of templates, and suggests methods to control various problems you might encounter.

The Single-Copy Problem

There is a problem with using templates: you must ensure that, for each distinct set of parameters to a template, there is only a single copy of an instantiated function or static class data member. This is called the "single-copy problem."

The default operational mode of the High C/C++ compiler is to always generate instances of function templates and classes when they are used in a **source module - one instance** in the source module is generated per set of parameters to the template.

High C/C++ currently supports the following solutions to the single-copy problem:

- Merging of global template instantiations, also referred to as the COMDAT solution.

This is the default solution for template instantiation and virtual-function table generation when you use the SCDE linker. See “Using COMDAT Sections to Merge Global Instantiations” for a discussion of this solution.

- Controlling instantiations at compile time. This is an alternate method for template instantiation.

Use this solution if you can not link with the SCDE linker. Specify option -**Hnocomdat** when you use this solution.

Using COMDAT Sections to Merge Global Instantiations

The COMDAT solution is the default solution for template instantiation and virtual function table generation. If you plan to use a third-party linker, you should adopt the manual instantiation techniques outlined in this section.

Limitations of COMDAT

COMDAT does not allow you to do the following:

1. Declare apocryphal some class template T in Module 1 and provide definitions for its member functions; then
2. Declare class template T in Module 2 without member function definitions, and instantiate template T in Module 2.

Instantiating Templates Manually

This section, which describes how to use compiler options and pragmas to manually control instantiation of templates, contains the following sections:

- Overview
- Preventing Duplicate Function Definitions
- Selective Template Instantiation
- Checking Template Usage
- Problems with Matching const Parameters

Overview

Note: If you can not use the SCDE linker, you must manually instantiate templates as described in this appendix, in order to avoid the single-copy problem. Automatic instantiation using COMDAT information requires special non-standard linker support.

You must specify option **-Hnocomdat** when you use this manual solution. For background information about the single-copy problem, see “The Single-Copy Problem” earlier in this chapter.

Locally Defined Templates

If you are using templates defined locally within a single source module, there is no single-copy problem and no special care is needed. Each source module has its own templates and instances, as shown here:

File *1.cpp*:

```
template <class T> T min(T x, T y) {...}
...
int i = min(1,2) + min(3,4);
```

File *2.cpp*:

```
template <class T> T max(T x, T y) {...}
...
int j = max(1,2) + max(3,4);
```

In this example, each module contains an instance of a template function: min for *1.cpp* and max for *2.cpp*.

Duplicate Definitions

The single-copy problem arises only when you are using the same template in separate source modules, as shown here:

File *hdr.h*:

```
template < class T > T min(T x, T y) {...}
template < class T > T max(T x, T y) {...}
```

File *1.cpp*:

```
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

File *2.cpp*:

```
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

In this example, the object modules for *1.cpp* and *2.cpp* both contain definitions of `min(int, int)` and `max(int, int)`, and the linker will complain about a duplicate definition.

Preventing Duplicate Function Definitions

One way to prevent duplicate definitions is to turn off automatic template instantiations when compiling. You turn them off for template functions by turning Off toggle *Auto_func_instantiation* as shown here:

File *1.cpp*:

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int i = min(1,2) + max(3,4);
```

File *2.cpp*:

```
#pragma Off(Auto_func_instantiation)
#include "hdr.h"
int j = min(1,2) + max(3,4);
```

The toggle must be turned Off in the definition of the function template - not at the reference to the function template.

Essentially, the compiler records the setting of the toggle when it encounters the function template. For a non-inline function template encountered with the toggle Off, there are no instantiations of the function template bodies in the source code.

But now you have two modules, both of which make external references to `min(int, int)` and `max(int, int)`, and the linker will complain about missing functions.

You can provide the necessary functions as follows. In a single new source module, include calls (with the appropriate arguments) to the function templates, and keep the *Auto_func_instantiation* toggle On. The object code for this new module contains the required instantiated functions.

We recommend you place these calls to the function templates in the module in such a way that the functions are never actually called at run time, and the code for calling them is never actually generated by the compiler.

The following example shows a convenient way to do this with macros:

File *fti.cpp*:

```
#include "hdr.h"
#define DONTCALL(f) (0 && f)
static void dummy() {
    DONTCALL(min(1,1));
    DONTCALL(max(1,1));
};
```

Because of the conditional expression in the macro, the body for `dummy()` is empty. Yet the object module for this source module contains the definitions of `min(int, int)` and `max(int, int)`.

Class Templates and Duplicate Definitions

A class template must always be instantiated when it is used because the resultant type is required for further processing but the single-copy problem applies to the non-inline class member functions and static data members. See “The Single-Copy Problem” for more information.

If you use a class template only locally to a source module there is no problem; the problem arises when you share an instantiation of a member function or data member.

As with function templates, the compiler by default always generate instances of class members when the class is used in a source module, and also when the definitions of those members are presented to the compiler. You can omit the definitions if you want.

You can turn off this automatic instantiation by turning Off toggle *Auto_class_member_instantiation*. Then you must provide another source module (with the toggle On), in which you define the class members, as shown in the following example:

Turning Off Automatic Instantiation

File *stack.h*:

```
template < class T > class stack {
    T *v, *p; int sz;
public:
    stack(int);
    ~stack() { delete[] v; }
    void push(T);
    T pop() { return *--p; }
};
template<class T> void stack<T>::push(T a) {
    *p++ = a;
}
template<class T>      stack<T>::stack(int s) {
    v = p = new T[sz=s];
}
```

File *3.cpp*:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
    stack<int> I(10);
    stack<float> F(20);
    I.push(1);      F.push(1.0);
    I.pop();        F.pop();
}
```

File *4.cpp*:

```
#pragma Off(Auto_class_member_instantiation)
#include "stack.h"
f() {
```



```

stack<int> I(10);
stack<float> F(20);
I.push(2);      F.push(2.0);
I.pop();        F.pop();
}

```

In this example, both modules contain external references to the following:

```

stack<int>::push(int)
stack<int>::stack(int)
stack<float>::push(float)
stack<float>::stack(float)

```

Note that there are no references to the inline members (such as `pop`) of **stack<int>** and **stack<float>**. These inline members are in fact generated by the compiler and inlined. The external references are made only to the non-inline functions and static data members.

You can provide the `push( )` and constructor functions by compiling the following module, which simply refers to the two stack types:

File *cti.cpp*:

```

#include "stack.h"
typedef stack<int>    dummy1;
typedef stack<float> dummy2;

```

Selective Template Instantiation

Regardless of how toggle *Auto_class_member_instantiation* is set, you can ensure or suppress the instantiation of selected template types with pragmas **Ensure_instantiation(T1, T2,...Tn)** and **Exclude_instantiation(T1, T2,...Tn)**, where *T1, T2...Tn* are names of specific template types.

Pragma **Ensure_instantiation**

When you use pragma **Ensure_instantiation**, the class members for each template type in its argument list are instantiated. Here is an example:

```

#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
    int f();
};
template <class T> Q<T>::f() {
    return sizeof(T);
}
typedef Q<int> x;

```

```

typedef Q<double> y;
typedef Q<Q<int> > z;
// Make sure Q<double> and Q<Q<int> > are
// instantiated
#pragma Ensure_instantiation(Q<double>, Q<Q<int> >)
// Q<int> will NOT be instantiated, because it is not
// listed

```

Pragma Exclude_instantiation

When you use pragma **Exclude_instantiation**, you suppress instantiation of the class members for each template type in its argument list. You can use pragma **Exclude_instantiation** to resolve the single-copy problem in programs compiled from two or more source modules. If you find, after linking the program, that a template has been instantiated in two or more object files, add pragma **Exclude_instantiation** to all but one of the source files and recompile.

In the following example, file *1.cc* and file *2.cc* both contain an instantiation of **Q<int>**:

File *x.h*:

```

template <class T> struct Q {
    int f();
}
template <class T> Q<T>::f() {
    ...
}

```

File *1.cc*:

```

#include "x.h"
Q<int> x;

```

File *2.cc*:

```

#include "x.h"
Q<int> y;

```

To resolve the problem, modify file *2.cc* as follows, and recompile:

Modified file *2.cc*:

```

#include "x.h"
Q<int> y;
#pragma Exclude_instantiation(Q<int>)

```

Forcing Instantiation of Template Types

You can use toggle *Ensure_instantiation* to force instantiation of template types. Any template used within the scope of this toggle gets instantiated.

For example:

```
#pragma Off(Auto_class_member_instantiation)
template <class T> struct Q {
    int f();
}
template <class T> Q<T>::f() {
    return sizeof(T);
}
typedef Q<int> x;
typedef Q<double> y;
#pragma On(Ensure_instantiation)
// Q<int> and Q<Q<int> > will both be instantiated
typedef Q<Q<int> > z;
#pragma Off(Ensure_instantiation)
```

With toggle *Ensure_instantiation*, it is harder to predict exactly what will be instantiated than with pragma **Ensure_instantiation**, because one template usage can indirectly cause other templates to be used. In the last example, the use of **Q<int>** causes the expansion of **Q**'s class body, in which any other template types used are then instantiated.

Checking Template Usage

To get an idea of which templates were used in a compilation, turn On toggle *Print_template_usage*. This toggle prints both function templates and class templates. It also prints the members of a class template, indented below the class template. A single character in the leftmost column indicates the following:

It means the compiler instantiated this. **e** means undefined (it is external) and you must supply it. **U** means the user defined a special version of a function, class member, or class template

The following output shows the result of compiling *3.cpp* (see Example above):

Template usage:

```
I    stack<int>                                (at "3.cpp", L5/C7)
```

```

e      stack(int)           (at "stack.h",L4/C11)
e      void push(int)       (at "stack.h",L6/C16)
I      stack<float>         (at "3.cpp",L5/C25)
e      stack(int)           (at "stack.h",L4/C11)
e      void push(float)     (at "stack.h",L6/C16)

```

This output shows that the two stack classes are instantiated (I) but the two member functions are external (e).

When you compile *cti.cpp* (see the discussion following previous example), you see the two stack classes instantiated:

Template Usage

```

I      stack<int>           (at "cti.cpp",L2/C11)
I      stack(int s)         (at "stack.h",L10/C26)
I      void push(int a)     (at "stack.h",L9/C26)
I      stack<float>         (at "cti.cpp",L3/C11)
I      stack(int s)         (at "stack.h",L10/C26)
I      void push(float a)   (at "stack.h",L9/C26)

```

Problems with Matching const Parameters

Instantiating a function template requires an exact match of the function formal-parameter types and the actual-parameter types. This requirement can be a nuisance.

For example, suppose you generalize the ANSI Standard C `bsearch()` routine with a template:

```

template <class T> T *bsearch(
    const T key,
    const T *base,
    int nmemb);

```

Because you used `const`, your generic search routine “promises” never to clobber the elements being searched, nor the key. This is a valuable promise. But when you try to call `bsearch()`:

```

extern int a[10];
int *solution = bsearch(123, a, 10);

```

the call fails, because `123` is not a `const`, and `a` is not of type pointer-to-`const`. In fact, to successfully call `bsearch()`, you must write this:

```

extern int a[10];

```

```
int *solution = bsearch(
    (const int)123,
    (const int *) a, 10);
```

One way to alleviate this problem is to define an additional inline template that allows non-const parameters, but which calls the “real” one:

```
template <class T> inline T *bsearch(
    T key,
    T *base,
    int nmemb) {
    bsearch(
        (const T)key,
        (const T*)base,
        nmemb);
}
```

This template allows the call to `bsearch(123, a, 10)` to succeed. The second template is called, which turns around and calls the first. Because the second template is inlined, there is no performance penalty.

To permit a non-`const` parameter to be “matched” with a `const` parameter in a function template, turn On toggle *Template_trivial_conversions*. Turning this toggle On allows trivial conversions (as defined in the Annotated C++ Reference Manual) when arguments are matched to function templates. This conversion allows the call to `bsearch(123, a, 10)` to work without introducing the intermediate inline template.

Diagnostic Messages

This appendix contains the following information about diagnostic messages:

- File I/O Errors
- System Errors
- User Errors and Warnings
- Warning Levels
- Error and Warning Messages
- Changing the Message Format
- Controlling Warning Messages

File I/O Errors

File I/O errors are fatal. They can occur when you try to open a non-existent file, or when the compiler writes an output file and not enough file space is available. The I/O error messages you are most likely to encounter are:

- `fff: file not found.`

This message is produced when a source file specified on the command line cannot be found. In addition, when running the compiler in ANSI mode, the files *hcansi.st*, *hcansi.pt*, and *hcansip.pt* must be accessible. These files typically reside in */usr/ccs/bin*.

- `***Cannot write to intermediate file.`

This message is usually caused by too little space on disk. Remove unnecessary disk files and try again.

Fatal errors can result in compiler temporary files being left on the disk. These are files with a *.tmp* suffix. They should be removed.

System Errors

System errors are fatal and you should rarely encounter them. The error messages take the form:

```
>>>> S Y S T E M E R R O R n <<<<, in Module:Function
Error message text.
```

where *n* numbers the occurrences of system errors, *Module* is the module name, and **Function** is the function name. The only system error messages with which the user should be concerned are as follows:

- **Dynamic array allocation/reallocation failed. Out of memory.**
This error indicates that there was too little memory to compile the program. This usually occurs when at least one function definition is pathologically large. Such functions should be broken up into smaller ones.
- **Recover: Exceeded the following limit: Limit.**
A table overflowed while a syntax error was being repaired. The table limit is fixed, so no increase in memory can improve the situation. Repair the error.

Other system error messages are associated with internal compiler errors or inconsistencies that normally should not occur. If they do occur, contact your service representative.

User Errors and Warnings

User error messages are grouped in three categories:

- Lexical Error Messages
- Syntactic Error Messages
- Constraint Error and Warning Messages

Warnings do not suppress object-file generation; errors always do. Also, some diagnostics that are warnings become errors when the compiler is run in ANSI mode.

Messages that report errors terminate compilation after the phase issuing the diagnostic, so errors that would otherwise have been detected by later phases are not reported until all earlier errors are repaired and the compiler is reinvoked.

All user diagnostics are accompanied by the file name, a line number *n*, and column number *m*, in the form *filename*, *Ln/Cn*, reporting where the error was detected. In addition, lexical and syntactic errors are generally accompanied by the erroneous line with a caret (^) beneath it at the point of error detection. Error messages begin with **E** and warnings with **w**, and usually occupy a single line.

Lexical Error Messages

Lexical error messages are produced when an improperly formed word is detected, such as a string with a missing closing quote.

Example

```

Levels      LINE #      | ----+----1----+----2----+----3----+----4-
---+-----5
          1      |void main() {
1      1      2      |      char *S;
1      1      3      |      S = "Hello;
                  C15  -----^
E "file", L3/C15: (lexical) Unexpected end-of-line encountered.
1 1      4      |      }
```

Syntactic Error Messages

Syntactic error messages are produced for programs that are ill formed at the phrase level, such as a missing semi-colon (;) or inserted spurious symbol. The message is accompanied by a statement of the REPAIR that the compiler effected so it could keep processing input.

Example

```

Levels      LINE #      |----+----1----+----2----+----3----+----4--
---+-----5
1          1          1  |void main() {
1          1          2  |      printf "Hello");
C11 -----^
1          1          3  |      }
E "file", L2/C11: (syntactic) unexpected symbol:'<S TRING>':"He
llo"
REPAIR: '(' was inserted before '<STRING>':"Hello"@ L2/C11

```

Constraint Error and Warning Messages

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are more than 200 such diagnostics, each of which is meant to be self-explanatory. Most of them prevent the generation of object code, but some are merely warnings intended to assist the programmer.

Examples

```

Levels      LINE #      |----+----1----+----2----+----3----+----4--
---+-----5
1          1          1  |void main() {
1          1          2  |      int i;
1          1          3  |      i = Undeclared_identifier ;
1          1          4  |      }
E "file", L3/C8: Undeclared_identifier: This is undeclared.
1 user error No warnings
Levels      LINE #      |----+----1----+----2----+----3----+----4--
---+-----5
1          1          1  |void main() {
1          1          2  |      int i, Unused;
1          1          3  |      i /= 0;
1          1          4  |      }

```

User Errors and Warnings

```
w "file", L2/C8: i: Variable is set but is never referenced.  
w "file", L2/C11: Unused: Variable is never used.  
E "file", L3/C6: Division by zero.  
1 user error 2 warnings
```

Warning Levels

The compiler supports four levels of warning diagnostics. Level 1 warnings are considered to be of the most interest to the programmer. Level 2 and 3 are less so; level 4 is merely informational.

The warning level can be set from the command line with the **-wn** option, or set in source with the **Warning_level** pragma. For a description of the **-wn** option, see pragma **Warning_level** in the “Pragma Syntax” section of Chapter 3, “Compiler Pragma.” The **Warning_level** pragma has the following form:

```
#pragma Warning_level(n)
```

where *n* is a decimal integer constant denoting the level. Setting the warning level to *n* means that warning diagnostics with severity level *n* or less are displayed. A setting of 0 suppresses all warnings. The default warning level is 3.

Messages and classifications can change as the compiler is revised. As of this writing, warning messages are classified as follows:

Examples

Level 4

Old-style K&R C would possibly produce different results for relational.

Level 3:

- '=' encountered where '==' may have been intended.
- Function called but not defined.
- Unsigned compare with zero always false/true.
- Variable “name” is never referenced.

Level 2:

- Expression has no side-effects.
- Prototype causes non-standard conversion from “type1” to “type2”.
- (Also under control of `Prototype_conversion_warn` toggle.)
- Static function is not referenced.
- Toggle “name” is unrecognized.
- Variable “name” is possibly referenced before set.

All other warning messages are Level 1.

Error and Warning Messages

This section presents all compiler diagnostic messages (except automatically generated lexical and syntactic messages) in alphabetical order, with explanations where appropriate.

- ‘)’ missing in macro call.
- #define directive is malformed.
- #else or #elif encountered within a pending #else.
- #pragma statement is malformed.
- ‘=’ encountered where ‘==’ may have been intended.
= was detected as an operator in a Boolean expression, such as if (x = y) {...}. Often this is a mistake, as if (x == y) {...} was intended.
- Anonymous structure member is ambiguous.
- Applying “offsetof” to a non-struct type.
- Argument name is missing in function declaration.
For function definitions, argument names must be supplied. For example, void f(int,float g){...} is illegal because the first argument lacks a name.
- Argument name is not specified in function header.
For function definitions, arguments declared must be present in the function header. For example, void f(a,b) int x;...; {...} is illegal because x is not present in the function header.
- Argument number n not named.
The nth argument in a named call is not specified.
For example:

```
int f ( int a, int, int c ) {...}
...
f(a=>1, c=>2);
```

The second argument is not specified.
- Argument of type void not permitted.
- Argument of unknown length being passed.

- Argument type not consistent with previous specification.
- Array dimension is zero or negative.
- Array element type is a yet undefined struct/union.
- Array subscript out of range.
- Assignment into a const variable is attempted.
- Assignment into a variable of unknown length is attempted.
- Assignment into something of type void is attempted.
- Attempting to apply the function abs to an unsigned quantity.
- Attempting to define a variable in data section name that was previously defined at Ln/Cn.
- Attempting to take the difference of two pointers pointing to objects of unknown length.
- auto must appear within a function. static assumed.
Storage class auto cannot be given for declarations that do not appear within a function.
- Bad line specifier in #line directive.
- Bit field exceeds word size. Bit field is not an integer type.
- Bit field is not valid as an argument to “sizeof.”
Since bit fields need not occupy an integral number of bytes, taking their sizeof is prohibited.
- Bit field value expected but encountered ‘{’.
- “break” statement not within loop or switch.
- Call to a function returning an incomplete or zero-length type is attempted.
A function cannot return a struct or union type whose fields have not yet been specified. For example, struct s *f(){...} is legal since f returns a pointer to an incomplete struct type, but struct s g(){...} is illegal.
- Call to non-function is attempted.
- Calling convention not consistent with declaration at xxx.

The calling convention being used at a function definition does not correspond to the calling convention that was designated at a previous declaration of the function (at location xxx) .

- Cannot dereference a pointer to void.

Type void* was introduced as a means of defining a “generic pointer” compatible with other pointers. But there is no such thing as an object of type void. Therefore, dereferencing a pointer to void is illegal.

- Cannot dereference pointer; size of object not known.
- Cannot return from iterator for-loop body.

As an implementation restriction, one cannot return from an iterator for-loop body; one can, however, break, continue, or use a goto inside an iterated for-loop body.

- Cannot take the address of a register variable.
- Cannot take the address of a struct bit-field.
- “case” encountered outside of a “switch” statement.
- Constant expression or static address is expected.
- continue statement not within loop.
- Data class specifier expected (EXPORT,COMMON,IMPORT).
- Data section class not consistent with previous specification at Ln/Cn.
- “default” encountered outside of a switch statement.
- “default” label previously specified in current switch.
- Dereferencing of a non-pointer is attempted.
- Division by zero is attempted.

This was detected in a constant expression at compile time.

- Duplicate case label (n); other occurrence at Ln/Cn.
- Duplicate declaration of identifier.
- Duplicate definition of structure member.
- End of file encountered within #if construct (unmatched #if-#else-#elif at Ln/Cn).
- enum tag is not defined.

A declaration such as enum x; was encountered. Tag x has no definition.

- enum tag previously declared as struct tag.
- enum tag previously declared as union tag.
- Expression has no side-effects.

An expression used in a statement context has no side effect; therefore the expression is useless. For example, `2+3;`.

- Expression of type “type” should be returned.
- Fewer iterator arguments specified than required.

Fewer values were provided for in the iterator-driven for-loop than were yielded.

- Floating-point constant expected.
- Function called but not defined.

Any function that was called but not defined is noted as a warning. Although this practice is permissible in C and is especially useful when calling library functions, a common error is to misspell a function name. Without this warning, the error goes undetected until link-time. Furthermore, errors in parameter linkage can occur when a call is made to an undefined function. We recommend that the library `.h` header files always be included to get parameter checking, and that function prototypes be used for external function declarations, rather than making use of the “feature” of C for calling undefined functions.

- Function declaration is inconsistent with previous declaration at Ln/Cn.
- Function declaration is inconsistent with the “int”-returning function declaration imputed at Ln/Cn.

A function called before it is declared is assumed to be a function returning `int`, and any subsequent declaration of the function must declare it to be so. For example, `main () { (...) f(3); (...) } void f() {...}` is illegal since `f` was called before being defined and therefore assumed to return `int`.

- Function definition semantics overridden by prototype at Ln/Cn.

A prototyped function declaration takes precedence over an ordinary function definition.

Example:

```
int f(int a, int b);
```

```

...
/* Prototyped functionality. */
int f (a,b)
int a,b; {...}

```

The function declaration in the function definition is overridden by the prototype declaration. This warning may be suppressed by turning off the *Prototype_override_warnings* toggle.

- Function has no return statement.
A function with a non-void return type contains no return statement. This typically happens with “old” C programs that did not use void to indicate that a function returns nothing.
- Function name missing.
- Function parameter names are allowed only on function definitions, not declarations.
int f(a,b,c); is a function declaration that names the parameters (a,b,c). This is illegal unless function prototype syntax is used, as in int f(int a, int b, int c);.
- Function specified with more arguments than its declaration allows.
- Identifier expected.
- Identifier is out of context.
- Identifier is undeclared.
- Identifier missing in declaration.
- Identifier missing in struct member declaration.
- Illegal construct within #if expression.
- Initialization entry is of wrong type.
- Initialization list is longer than aggregate being initialized.
- Initialization of a variable within a “COMMON” or “IMPORT” section is not permitted.
- Initializer type does not match type “char”.
- Initializing a function is attempted.
- Insufficient number of arguments.
- Insufficient number of arguments in pragma specification.
- Insufficient number of arguments to function.

- Insufficient number of arguments to iterator.
The actual number of arguments passed to an iterator must match the formal argument list.
- Insufficient number of arguments to macro.
The number of arguments to a macro must agree exactly with the number of parameters in its #define.
- Integer constant exceeds largest unsigned number.
- Integer constant expected.
- Invalid call to a parpar function from pipeline.
- Invalid call to a pipepar function from parallel.
- Invalid call to a pipepipe function from parallel.
- Invalid use of “long” adjective.
- Invalid use of “short” adjective.
- Invalid use of type adjective.
- Iterator call required.
An iterator was called that was not defined.
- Iterator functions must be of type void.
Iterators cannot yield a value through the return statement. Thus, all iterators must have return type of void.
- Label identifier happens to have the same name as an enum identifier.
Perhaps a preceding “case” is missing.
The compiler issues this warning for label B in the following program segment:

```
enum {A, B, C} i;
...
switch(i){
    case A: B: ... /* Meant to say "case B:" */
    case C: ...
}
```

The user most likely intended the keyword case to precede the label “B:”.

- Label was not defined.
- Label was previously defined at Ln/Cn.

- Left side of '=' is an array, which is not an lvalue.
In this context a so-called "lvalue" is required but was not found. An lvalue is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to &, ++, and --. An array is not an example of an lvalue.
- "long char" treated as "short int".
- "long float" treated as "double".
- Lower bound of range greater than upper bound.
This can only happen in High C case statements where range expressions are allowed as labels (an extension).
- Macro call has too many arguments.
- Macro expansion nested too deeply.
- Malformed conditional compilation control sequence.
- May not cast a nested function to type "type".
- May not have functions as array elements (but may contain pointers thereto).
- May not return a function from another function (but may return a pointer thereto).
- May not return an array from a function (but may return a pointer thereto).
- Member not defined in referenced structure.
- Mismatched #if-#elif-#else-#endif.
- Missing macro argument after or before "##" operator.
- Missing macro argument after "#" operator.
- More iterator arguments specified than required.
More values were provided for in the iterator-driven for-loop than were yielded.
- Name is not a member in the applicable structure.
- Name is undefined; "pragma Alias" ignored.
- Named bit-field of length 0 not permitted.

A declaration such as `struct { int i:0, j:2 };` was encountered. `i` must be omitted. As is, it is possible to refer to the field. Such a reference would be illegal.

- Nested function may not be assigned into a pointer to a function.
- No “pragma Data” is active.
 pragma Data; was encountered without a preceding, and matching, **pragma Data(...);**.
- Not a variable or function; “pragma Alias” ignored.
- “offsetof” operation cannot be applied to bit field.
- Old-style K&R C would possibly produce different results for relational.
- Operand must be of an integer or float type (instead of type).
- Operand of ++ or -- is a pointer to an object of unknown length.
- Operand of the function “abs” must be of arithmetic type.
- Operand of type “type” is not valid for operator operator.
- Operands of type “type1” and/or “type2” are not valid for operator operator.
- Packed structs are not supported; “packed” ignored.
- Packed structs are not supported; status of `ALIGN_MEMBERS` toggle ignored.
- Parameter not found or specified more than once.
 In a function call using named parameter association, a parameter was named twice, or a non-existent parameter was referenced.
- Parameter “name” not supplied.
 Named parameter not specified.

Example:

```
void P ( int a, int b, float c ) {...}
...
P (a=>1, c=> 3);
```

does not initialize the named parameter `b`.

- Passing an argument of type `type1` where `type2` is expected.

An attempt was made to pass an argument of a wrong type to a function such as passing a float for a parameter that is a struct.

- Pointer to an object of unknown length is being indexed.

Attempting to index a pointer to an object of unknown length. `struct x *p; *p[0]` is an example of this type of error.

- “pragma Code” specified within function definition; ignored.
- “pragma Data” active at end of module.

A “pragma Data (...);” was given in a module or function, with no terminating “pragma Data;”. This is permitted but the programmer may have forgotten to supply the terminating pragma, thus perhaps including more data declarations in a data segment than intended.

- Previous “pragma Data” is still active.
“pragma Data (...);” was given in the context of an already active “pragma Data (...);”. Insert “pragma Data();” preceding the offending pragma to “turn off” the active pragma.

- Previous definition of macro superseded.
Redefinition of a macro is permitted. The redefined macro takes precedence over existing definition of the macro.

- Prototype causes non-standard conversion from “type1” to “type2”.
The prototype syntax, causes the non-standard conversion from type1 to type2. This is signaled as a warning to indicate to the user that such a conversion is taking place.

- Real constant has too many digits.
- Redundant class specification.
- “register” must appear within a function. “static” assumed.
Storage class register cannot be given for declarations that do not appear within a function definition.

- Right operand of shift operator exceeds word size (=n).
- Right operand of shift operator is negative.
- Right operand of “%=” operator is zero.
- Section name not specified in “pragma Data;” pragma ignored.

- Selection of a member from a non-structure is attempted.
- “short char” is not supported.
- sizeof being applied to “void” type.
- sizeof being applied to a function type.
- sizeof being applied to an array of unknown length.
- sizeof being applied to incomplete struct/union.

The sizeof a struct or union type whose fields have not yet been specified is not known. For example, struct s; (...) sizeof (struct s) (...) is illegal because the size of the structure is unknown.

- Specifier “class” ignored; no variable being declared.
In a declaration such as “static struct s{ int x;}”, the storage class static is useless since no object was declared.
- Specifier “specifier” is not valid for a formal argument.
- Static function is not defined nor referenced.
A function of storage class static is neither defined nor called anywhere in the compilation unit.
- Static function is not defined.
- Static function is not referenced.
A function of storage class static is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function and it is essentially deleted.
- Storage class “class” is not valid for a struct member.
- Storage class “class” cannot be initialized.
- Storage class for function should be static or extern.
- String constant denoting section name expected.
- String exceeds the length of array being initialized.
- String exceeds the length of the array being initialized.
- String expected.
- String truncation required.
- struct tag was previously declared as a union.
- struct was previously defined as a union.

- struct/union tag was previously declared as an enum.
- struct/union tag was previously declared as an enum.
- Structure has no contents (is of size zero).
- Superfluous “pragma Code” specifiers.
- Superfluous “pragma Data” specifiers.
- Superfluous type qualifier.
- Superfluous type specification.
- switch expression cannot possibly have the value n.
- switch statement has no cases.
- Symbol declaration is inconsistent with a previous declaration at Ln/Cn.
- Symbol is predefined as an intrinsic.
- Tag name is being truncated to n characters in length.
- The rest of this line is extraneous.
- Toggle “name” is unrecognized.
- Toggle not recognizable.
- Toggle specifier must be an identifier.
- Too many arguments in “pragma Alias”.
- Too many arguments specified.
- Type “type” is not appropriate for a conditional expression.
- Type “type1” is not assignment compatible with type “type2”.
In an assignment expression, the right operand of type type1 may not be assigned to the left operand of type type2.

In a function call, an argument of the type type1 may not be passed to a function that expects a parameter of type type2.
- Type “type1” is not compatible with the type “type2”.
In a comparison, the left operand of type type1 may not be compared with the right operand of type type2.
- Type “type” is not valid for a struct member.
- Type of entity is a yet undefined struct/union.
- Type of entity is an array of unknown length.

- Typedef name may not be qualified with adjective.
- Unable to assign function address; calling convention mismatch.
- Unable to cast type “type1” to type “type2”.
- Unable to determine the size of array type.
- Unable to initialize array; element type not complete.
- Undefined structure member.
- Unexpected character “c” follows “\”.
- Unexpected symbol in expression. Line ignored.
- union tag was previously declared as a struct.
- union was previously defined as a struct.
- Unreachable statement.
- Unrecognizable data section class.
- Unrecognizable pragma directive or toggle: name(text).
- Unrecognizable pragma directive: name.
- Unrecognizable pragma.
- Unrecognized preprocessor directive: “name”.
- Unsigned compare with zero always false.
- Unsigned compare with zero always true.

An expression of type unsigned int is never less than zero. Thus, a comparison to check whether the unsigned int expression is greater than zero is always true, and a comparison to check if it is less than zero is always false.

- Up-level reference to a register-class variable is not allowed.
- Value of escape sequence exceeds maximum value representable in unsigned char.
- Variable “name” is never referenced.
- Variable “name” is possibly referenced before set.

This warning is issued by the optimizer when it has found an auto- or register-class variable that is “live” at its declaration. In other words, the compiler has detected a potential path in which such a variable is referenced prior to being assigned a value.

This condition is not necessarily an error in that the “potential” path might never occur. For example, the variable “x” in the following code fragment would be diagnosed with this warning:

```
static void foo(int i) {
    int x;
    if (i >= 0) x = 1;
    if (i < 0) x = 2;
    printf("x=%d\n", x);
}
```

- Variable expected.

In this context a so-called “lvalue” is required but was not found. An lvalue is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to `&`, `++`, and `--`. The rules of C require the automatic conversion of some objects into non-lvalues. For example, the operand of `&` must be an lvalue, so `int i = &(a+b)` produces the “Variable expected.” diagnostic. A common cause of this message is the use of a construct such as:

```
int * p;
c = * (( char*)p)++;
```

which is legal on most PCC-compatible compilers, but disallowed by the ANSI Standard. Use instead:

```
int * p ;
c = *(* (char**) &p)++;
```

to circumvent the restriction. This solution works only when `p` is not a register variable; unfortunately we know of no solution for register variables.

- Variable previously initialized at Ln/Cn.
- Variables within a user-specified data section must not be “extern”.
- Variables within an imported data section must be “static”.
- “void” is not a valid argument type.
- “volatile” qualifier in cast has no effect.

The warning is issued to remind the user that simply casting a variable to a volatile type does not force the variable to be volatile. This is because the variable being cast is actually an rvalue. For example, the cast in the following assignment accomplishes nothing:

```
int a,b;  
...  
a = (volatile int)b;
```

The effect of forcing `b` to be volatile can be attained by the following sequence:

```
a = *(volatile int *)&b;
```

Changing the Message Format

You can modify the format of error messages and warning messages with command-line options **-Hefmat** and **-Hwfm**, respectively. These options direct the compiler to format error and warning messages according to a quoted string containing a sequence of characters with special meanings. (You can look at the .cnf file to view the actual text of the format directive.)

The formatting characters include the following:

%c	Column number
%f	File name
%l	Line number
%m	Comma: direct the compiler to include a comma
%n	Diagnostic message number
%q	Double quote: use to avoid quoting-convention problems of some shells
%t	Tab

Predefined Message Formats

You can specify several predefined formats by using special command-line options. Suppose this is the default High C/C++ format of an error message:

```
E "test.c",L1/C7/#237: J: Identifier is undeclared.
```

This same message would appear by default.

Controlling Warning Messages

You can control many of the warning messages output by the compiler by level or by message number. Warning messages are output by default. You can turn specific numbered warning messages off with pragma **Offwarn**, described in Chapter 3, “Compiler Pragmas”.

Controlling Groups of Warnings by Level

The compiler supports five levels of warning diagnostics.

- Level 1 warnings are considered to be of the most interest.
- Levels 2 and 3 warnings are of less interest.
- Level 4 warnings are informational.
- Warning level 0 suppresses all warnings.

You can specify the level of warning messages to display with option **-wn** or with pragma **Warning_level(n)**. The **Warning_level** pragma has the following form:

```
#pragma Warning_level(n)
```

where n is a decimal integer constant denoting the level. Setting the warning level to n means that warning diagnostics with severity level n or lower are displayed. A setting of 0 (zero) suppresses all warnings. The default warning level is 3.

For more information, see:

- option **-wn**, described in Chapter 2, “Invoking the Compiler”
- pragma **Warning_level**, described in Chapter 3, “Compiler Pragmas”

Controlling Individual Warnings by Number

Many warnings can be controlled individually by message number. The following pragmas override the current level of warning messages set by pragma **Warning_level(n)** or command-line option **-wn**:

- **Onwarn**
- **Onwarn_error**
- **Offwarn**
- **Popwarn**

For more information about these pragmas, see Chapter 3, “Compiler Pragmas”.

Suppressing Individual Warnings

To suppress individual warnings, first find the warning numbers by compiling with toggle *Msgno* turned On (the default). Then use pragma **Offwarn** to specify the numbers of the warnings you want to suppress. For example, this warning (#257):

‘=’ encountered where ‘==’ may have been intended.

has been disabled in this code:

```
#pragma Offwarn(257)
... if (i = j) ...    /* Warning disabled    */
#pragma Popwarn()
... if (j = k) ...    /* Warning restored    */
```

Restoring Warnings

You can use pragma **Onwarn** to restore (re-enable) the warnings, although pragma **Popwarn** is preferable. When you want the compiler to be more stringent than usual, but you do not want to use full ANSI mode, use pragma **Onwarn_error**.

Changing Warnings to Errors

To change warnings to errors, find the warning numbers by compiling with toggle *Msgno* turned On (the default). Then use pragma **Onwarn_error** to specify the numbers of the warnings you want to promote to compiler errors.

For example, the compiler emits warning #60 when you use `return;` within a function that returns `int`. You can change this warning to an error, and cause the compilation to fail, by including this pragma:

```
#pragma Onwarn_error(60)
```

in your code.

However, not all warnings are emitted at the point in the code where the offending text applies. For instance, functions called but not declared are diagnosed at the end of compilation, so that only the last warning pragma has effect for all of them. For example:

```
#pragma Offwarn(291)
... undef_func1() .../* No warning issued here */
#pragma Popwarn()
... undef_func2() ... /* Nor here */
/* Now all undefined function warnings are
   emitted, so that both lines are diagnosed */
```

If your code contains undeclared functions, it is best to disable warning #291 for the entire compilation. You can use **-Hpragma** to specify this on the command line:

```
cc prog.c -Hpragma=Offwarn(291)
```

For more information, see:

- options **-Hpragma** and **-wn**, described in Chapter 2, “Invoking the Compiler”
- toggle *Msgno*, described in Chapter 4, “Compiler Toggles”
- pragma **Warning_level**, described in Chapter 3, “Compiler Pragas”

Making Cross-References

This appendix contains the following topics:

- Features of the Cross-Reference
- Using the hcxref Command
- Cross-Reference Pragmas
- Cross-Reference Format
- Distinction of File Names

Features of the Cross-Reference

Cross-references have the following features:

- Source-file line numbers
All cross-reference information refers to line numbers in the files compiled, as opposed to line numbers within a listing. Therefore no listing is necessary to use the cross-reference.
- Processing of include files
Included source files are handled properly. That is, they do not interfere with the process, and their names are included correctly in the results.
- Assignment distinguished from use
References that assign values into variables are distinguished from references that use values of variables.
- Optional annotated listing
It is possible to generate an annotated source listing of one or more program files. The listing contains cross-reference information to the right of the source text.
- Multiple modules
A cross-reference can span multiple compilation units by cross-referencing many modules at once and showing references from one module into the other. Thus, a single cross-reference can be produced for a program that is broken up into separately-compiled modules.
- Inter-module usage summaries
A list of the names that one module uses that are located in other files can be produced, organized by file. This can make it easier to understand the module interconnectivity of a large program.

Using the hcxref Command

The **hcxref** command processes one or more High C source files and produces a cross-reference listing on standard output. The listing consists of up to four components as described in the section “Cross-Reference Format.”

hcxref is a driver program that first invokes the compiler with option **-xref**, to generate cross-reference information from C source files, and then invokes program **xref** to format and display the information. If any of the source files contain errors, appropriate diagnostics are generated. **hcxref** invokes the compiler on each of the source modules, and invokes cross-reference processor **xref** on the concatenation of the result.

The command has the following form:

```
hcxref [options] [preprocessor_options]... files ...
```

where:

- options includes zero or more command-line options (listed below)
- preprocessor_options denotes zero or more preprocessor options (for example, **-Idir** or **-Dname**) that are required when compiling the files
- files denotes one or more High C source files

The options for **hcxref** are shown in the following table.

Table B-1. hcxref Options

Option	Purpose
-i	to expand include files in the listing. Ignored if used with -p
-l	to generate a listing of the source files, annotated with cross-reference information. Include files are not expanded in the listing unless -i is also specified. Ignored if used with -p
-m	For each module M, to produce a listing of the names referenced in M that are defined elsewhere
-p	Invoke an external C preprocessor for each source file, and process the output of the preprocessor instead of the source files themselves. If you do not specify -p, the internal preprocessor, which is part of the compiler, is used. The -l and -i options are ignored when used in conjunction with -p
-s	Generate statistics relating to the cross-reference
-u	Include in the cross-reference names that are declared but not referenced. These names do not appear in the cross-reference unless the -u option is specified

Cross-Reference Pragmas

You can use pragmas to control the cross-reference output. These pragmas must be specified in a profile file named *xref.pro*, which you must create and edit. The pragmas have the same form as High C compiler pragmas, except that strings are specified with single quotes (') instead of double quotes ("). The syntax is:

```
pragma Pragma_name(Xref_parameters);
```

Supported Pragma_Names

Supported *Pragma_names* are explained in this section.

On, Off, and Pop

These *Pragma_names* act identically to those for the compiler; see the chapter entitled “Compiler Toggles” in this guide. Supply one of the following toggles when you specify pragma **On**, **Off**, or **Pop**:

- *Annotate_includes*—Turn On this toggle to get an annotated listing of all files involved, not just files that are modules. This toggle is equivalent to the option **-i**.
- *Annotated_listing*—Turn On this toggle to get an annotated listing of all modules, but not “header” files. See the description of the annotated source listing below. This toggle is equivalent to the **-l** option.
- *List_module_usage*—Turn On this toggle to get a listing for each module M of the names used by M that are declared in other files. See the description of the module name list below. This toggle is equivalent to the **-m** option.
- *List_unused_includes*—Normally the cross-referencer does not list any names found in compiled include files that were never referenced. This default can be overridden by turning On toggle *List_unused_includes*. Also the cross-referencer never lists unused intrinsic names. That cannot be overridden. This toggle is equivalent to the **-u** option.

- *Statistics*—Turning On this toggle causes a summary of cross-referencer statistics to be produced at the end. This toggle is equivalent to the `-s` option.

Columns (Declared_name, Info, Refs, Right_margin)

The parameters are integer values that describe the format of the cross-referencer output:

- *Declared_name*—Column where declared names begin.
- *Info*—Column where related information begins.
- *Refs*—Column where references to declared names begin.
- *Right_margin*—Last column on page (determines page width).

The default is `Columns(12, 60, 90, 132)`.

Include ('File_name')

Here the parameter is a string denoting a file to be included. This works exactly like the **Include** pragma of High C except that the file name is enclosed in single instead of double quotes.

CAUTION: Do not confuse the **Include** pragma for the cross-referencer with that for the source language. They have the same effect—that of including source text—but the former is directed to the cross-referencer and the latter to the compiler. Files included in a compilation are referred to as “compiled include files” to avoid any confusion with files included in the input to the cross-referencer. (This does not include any files included with the “external” preprocessor, because the compiler has no knowledge of such includes.)

Example

```
File xref.pro:
pragma On(Annotated_listing);
pragma On(Statistics);
```

The following command generates an annotated listing and statistics for *my_prog.c*:

hcxref my_prog.c

Cross-Reference Format

Each cross-reference consists of four components: an object-name cross-reference, a file table (if more than one file is involved), a module name list (optional), and an annotated source listing (optional).

Object-Name Cross-Reference

The object-name cross-reference is an alphabetized list of all names declared in the program, together with an ordered list of all the references to each name. This list presents the following information for each distinct name in the program:

- the line and column number of the declaration of the name. If the name occurs in a compiled include file, or if several modules are being cross-referenced, the file number is also given.
- the declared name N, and its owner: the name of the function that contains N's declaration
- information about the named object, such as its storage class (such as static, extern, typedef, and register) and in some cases, the object's type
- the numbers of any lines containing references to the name. If the references are not in the module being cross-referenced (they may be in an included file), or if several modules are being cross-referenced, the line numbers are presented in the format fn<I>, where n is the number of the file containing the references and I is the list of line numbers. Occasionally the entry in this field is of the form resolved at ref where ref is a line number or fn<...> reference as just described. This means that the name was introduced by an extern declaration whose actual definition was given at ref.
- references that assign, or may assign, a value to a variable are marked with the asterisk character (*).

File Table

The file table is an alphabetized table of all files used in the program, with a file reference number for each. This table is produced if the cross-reference involves more than one file; this happens if more than one module is cross-referenced, or if any compiled include files were involved in the modules being cross-referenced. This table shows the correspondence between file numbers and file names. References in the object-name cross-reference and the list of module names use the file number rather than file names, to keep the listing brief. Use the file table to determine the corresponding file name.

Module Name List

A module name list is a list for each module *M* of all the names used by *M* that are defined in other files that are included in the cross-reference. This list is produced if the **-m** option is specified. The list is organized by file, and is useful for determining the interconnectivity between modules.

In the object-name cross-reference and the module name list a reference to a name *N* declared at reference point *P* is changed to a reference to a point *P'*, if the definition at *P'* resolves the declaration at *P*. Typically this happens when *N* is declared in an interface file *F*, is used in a module *M*, and is defined at *P'* in a module *M'*. The module usage in the module name list shows that *M* refers to *P'* in module *M'*, not *P* in interface file *F*. That is, you get references to the implementations rather than the interfaces through which they were supplied.

A module name list is generated only if the sharing of names between modules is done through interface files. Thus, for C, the modules:

```
m1.c:    int i1; extern int i2;
        main () { i1 = i2; sub(); }
m2.c:    int i2 = 4; extern int i1;
        sub () {printf("%d %d\n",i1,i2);}
```

do not share the names *i1* and *i2* because *i1* and *i2* are not declared in a common interface file. On the other hand, with:

```
glob.h:    extern int i1,i2;
m1.c:    #include "glob.h"
        int i1;
        main () { i1 = i2; sub(); }
```



```
m2.c:      #include "glob.h"
           int i2 = 4;
           sub () {printf("%d %d\n", i1, i2);}
```

the modules do share the names, and `i1` and `i2` are listed in the module name list. The cross-referencer is unable to determine that two modules refer to the same name unless they both refer to the name in the same interface file.

Annotated Source Listing

An annotated source listing is an annotated cross-reference for each module. This listing is produced if the `-l` option is specified. The result is a line-numbered listing of the source of the compiled program, with each line annotated on the right with the line numbers of the definitions of names used on the line.

If `n` names are used on the line, `n` line numbers appear to the right of the line, corresponding positionally. A line number alone is a reference into the file being listed. If the letter `i` appears instead, the name referenced is an intrinsic, such as `_find_char` or `_abs`.

Finally, a line number followed by `f` and another number means that the name was declared in a file other than the one being listed; the file number can be used to discover that file's name in the file table. For brevity, `Line#File#` is used instead of `File#<Line#>` as in the object-name cross-reference.

Distinction of File Names

In a multi-module cross-reference, a particular interface file might have been included by several modules, because each of the modules being cross-referenced needs the resources in that file. The cross-referencer assumes that a repeated declaration of a name in a compiled include file is the same declaration if it appears at the same line and column number of the same include file.

For purposes of determining “sameness of include files” the cross-referencer uses the text of the file name, including the path. Therefore, to cross-reference several modules successfully, do not use different names for the same include file.

For example, if module M1 includes *../utils/trees/trees.h* and M2 includes */home/myacct/utils/trees/trees.h*, and if these two references denote the same file, the cross-referencer does not recognize them as the same.

Optimizing Program Performance

This appendix covers the following optimization topics:

- Introduction to Optimization
- Specifying an Optimization Level
- Execution Speed Versus Code Size
- Level 0 Optimizations
- Level 1 Optimizations
- Level 2 Optimizations
- Level 3 Optimizations
- Level 5 Optimizations
- Level 6 Optimizations
- Level s Optimizations
- Cache Optimizations
- Processor-Specific Code Generation
- Basic Block Optimizing
- Additional Ways to Optimize Performance
- Function Call Count Reporting
- Structure Usage
- Intermediate Compilation

Introduction to Optimization

This chapter shows you ways to compile a program so that it runs faster or requires less memory. A speed improvement might come at the expense of code space, or the other way around. Some optimizations save both time and space. Many increase compile time.

You can activate some optimizations just by specifying the appropriate optimization level (see the section “Specifying an Optimization Level” later in this appendix) with command-line option **-on**. Others must be invoked with a separate command-line option, pragma, or toggle, instead of (or in addition to) specifying an optimization level (see the section “Additional Ways to Optimize Performance” later in this appendix).

Keep in mind that good coding practice can make a greater difference in code size or execution speed than any amount of compile-time optimization.

Specifying an Optimization Level

Optimization levels are:

- 0—for fastest compilation
- 1-7—for generating faster code
- s—for generating smaller code (SCDE Release 2.01 or later)
- c—for generating smaller, cache-packed code (SCDE Release 2.01 or later)

Each higher numbered level specifies additional optimizations, and includes the optimizations performed at lower levels with a few exceptions. (Preloading arguments from memory is performed only at level 1 and level s.)

Use command-line option **-On** (capital letter On) to specify the optimization level, where **n** is the level number (0 through 7) or the letter **s** or **c**. If you specify just **-O**, you get level 3 by default. If you do not use **-O** at all, level 3 is the default. In general, the higher the optimization level, the faster the generated code runs, usually at the expense of compile time. If compile time is more important to you than the speed of generated code, specify **-O0**.

Debugging Optimized Code

When you specify **-O0** or **-O1**, the compiler tries to generate code in source order. This makes your code easier to debug at the assembly-language level.

In general, optimizations make code harder to debug. Debuggers cannot usually determine when a variable's current value is in a register rather than in memory. Some of the effects that impact debugging are:

- Scheduling can cause erroneous line numbers to be reported for run-time errors.
- Inlining often prevents debuggers from stepping through a program correctly.
- During debugging, cross jumping can result in surprising jumps that are not reflected in the source code.

Execution Speed Versus Code Size

Most High C optimizations are intended to improve execution speed; some do so at the expense of code size. Some save code space, at the possible expense of execution time. A few save both time and space. You must decide which is more important to your application: faster execution or smaller code. If smaller code is more important to you, specify **-Os**.

The following table shows which optimizations affect execution time and which affect code space, and notes possible disadvantages. A primary benefit is one that almost always happens (optimizations sometimes backfire). A secondary benefit is one that less frequently makes a difference.

Table C-1. Optimization Benefits

T=time, s=space, c=compile time Optimization	Opt Lvl	Primary	Second- ary	Possible Disadvant- age
preload arguments from memory	1,s,c	S-	T-	
cross jumping	1,s,c	S-	T- ¹	T+ ²
iterative local CSE	2,s,c	T-,S-		
iterative live/dead analysis	2,s,c	T-,S-		
loop memory-reference elim.	2,s,c	T-,S-		
register lifetime analysis	2,s,c	T-,S-		
enhanced register allocation	2,s,c	T-,S-		
spill analysis	2	T-,S-		
global CSE	2	T-		S+
back-substitution of epilogue	2	T-		S+
low-level instruction scheduling	3,c	T-		C+ ³
improved spill analysis	3,s,c	T-,S-		
inlining	4	T-		S+,T+ ⁴ ,C+ ⁵

T=time, s=space, c=compile time Optimization	Opt Lvl	Primary	Second- ary	Possible Disadvant- age
pointer-based vars "behaved"	4	T-		
loop strength reduction ⁷	5	T-		
loop unrolling ⁷	6	T-		S+
optimize for space ⁷	s	S-		T+
tail recursion	n/a	T-	S-	
function prologue/epilogue	n/a	T-	S-	
function-calling conventions	n/a	T-	S-	
constants in code	n/a	T-	S-	
processor-specific instructions	n/a	T-		
floating-point optimizations	n/a	T-		
code alignment	n/a	T-		S+ ⁶
pack text ⁷	c			
limit footprint ⁷	c			
extended_frame_removal ⁷	3,c			
resolve 486 branch targets ⁷	3			

¹ Cross jumping can decrease time by decreasing the size of a function and therefore increasing its chances of being held in an instruction cache.

² Cross jumping can increase time by introducing extra branch instructions.

³ Generally, optimizations increase compile time, but with instruction scheduling compile time is noticeably increased.

⁴ Too much inlining can increase the size of a function beyond the ability to fit in a cache. Also, space may be increased unless the inlined function contains no more code than would be generated for the calling sequence that is replaced.

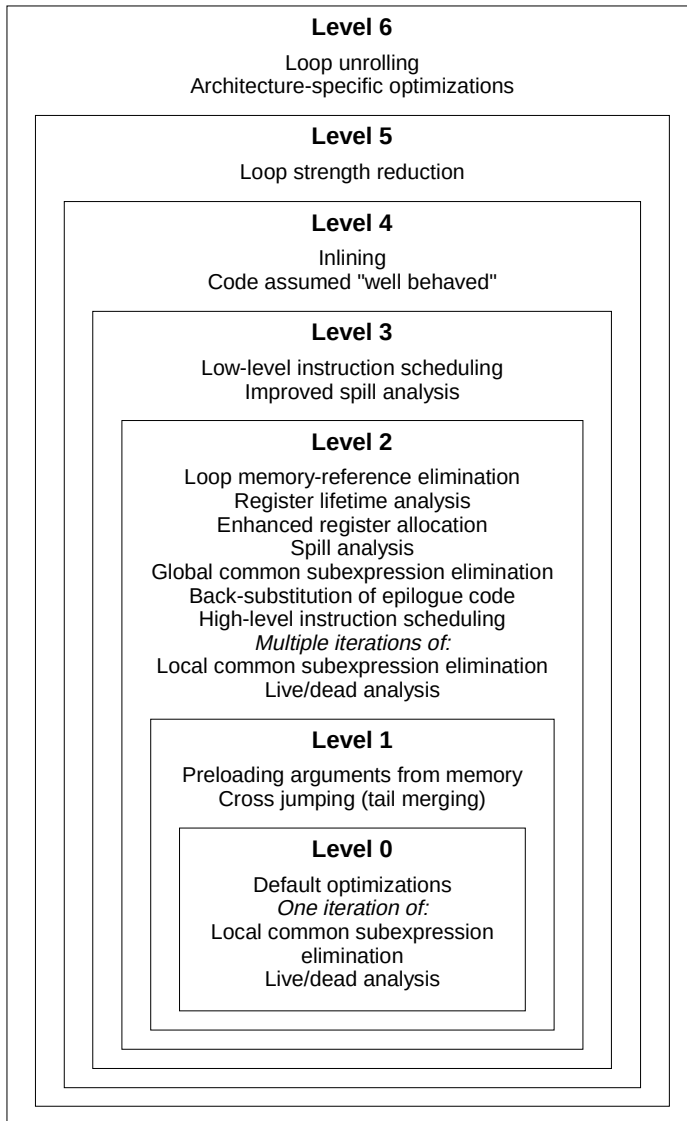
⁵ Inlining can make function definitions longer, resulting in increased compile time.

⁶ Whether there is much of a space trade-off depends on how many structures your application contains and how they are organized.

⁷ SCDE Release 2.01 or later

The following figure shows the optimization levels and associated performance. Levels 5 and 6 are in Release 2.01 or later.

Figure C-1. Optimization Levels



Level 0 Optimizations

This level includes default optimizations that you cannot disable. Level 0 also produces code in source order to make code easier to debug at the assembly-language level. Specify **-O0** if compilation speed is important to you.

Global Register Allocation

Operations using registers are often much faster than equivalent operations that reference memory locations. The compiler allocates registers to variables according to how frequently the variables are used. With global register allocation, the compiler may override explicit register declarations to achieve the most efficient usage.

Operator Strength Reductions

Sometimes an operation can be replaced by a more efficient operation (or series of operations) that yield the same result in less time. For example, multiplication by a constant can be converted to a series of additions, subtractions, and shifts. Division or modulo by an integer power of 2 can be converted to a shift or masking operation

Note: Some strength reductions, such as those that replace multiplication by additions and shifts, result in larger code size.

Local Common Subexpression Elimination

The compiler eliminates common subexpressions within extended blocks of code. An extended block is a section of self-contained code with one entry and one or more exits.

Constant Propagation

When the compiler sees a reference to a variable whose last assigned value was a constant, the reference is replaced by the constant.

Constant Expression Folding

Arithmetic expressions whose operands are constants are evaluated at compile time. Constant propagation often exposes such expressions that are not otherwise apparent.

Eliminating Unreachable “Dead” Code

The compiler eliminates code that cannot be reached by any execution path. Such “dead” code may be created as a result of another optimization. For example, constant propagation may have converted a conditional jump into an unconditional jump, eliminating a path out of the associated block.

Live/Dead Analysis

Computations whose results are never used are eliminated. Assignments of values that are never used (“dead stores”) are also eliminated.

Level 1 Optimizations

Level 1 produces code in source order as in level 0, and includes the optimizations described below.

Preloading Arguments from Memory

Controls: toggle *Preload_args_from_memory*

When possible, the compiler copies function arguments to machine registers in the function prologue to reduce memory fetches. Do not turn this toggle On when global common subexpression optimization is being performed. This toggle is Off by default at higher optimization levels.

Cross Jumping (Tail Merging)

Controls: toggle *Cross_jump*

Two or more basic blocks of code that end in the same sequence of code, and that have a common successor, can be rewritten to eliminate the redundant code. This is an effective space-saving optimization. The effect of cross-jumping on execution speed depends on matching characteristics and the nature of your application: it could slow down code, or speed it up, or it may have no effect on speed. You can disable cross jumping by turning Off toggle *Cross_jump*.

Level 2 Optimizations

Level 2 includes cross jumping from level 1, and the optimizations described below.

Iterative Local Common Subexpression Elimination

Controls: toggle *Local_CSE_iterate*

The compiler always performs local common subexpression elimination once. With this optimization, local CSE is performed iteratively to further improve the code. See “Local Common Subexpression Elimination” later in this appendix for a description.

Iterative Live/Dead Analysis

Controls: toggle *Live_dead_iterate*

The compiler always performs live/dead analysis once. With this optimization, the compiler performs live/dead analysis iteratively to further improve the code. See “Live/Dead Analysis” later in this appendix for a description.

Loop Memory Reference Elimination

Controls: toggle *Mem_refs_from_loop*

The compiler attempts to replace memory references to variables in loops with references to registers.

Register Lifetime Analysis

Controls: toggle *Loop_reg_rename*

Some disjointed loop variables are referenced in more than one loop but are “dead” upon exiting a loop. The compiler tries to allocate such variables to different registers in different loops.

Enhanced Register Allocation

Controls: toggle *Reg_alloc_enhance*

The compiler has a number of heuristics available to determine how to best allocate registers. By default, the compiler always uses one heuristic to allocate registers. With this optimization, all the heuristics are analyzed and the best one is used (SCDE Release 2.01 or later).

Spill Analysis

Controls: toggle *Reduce_reg_contention*

The register allocator attempts to limit the lifetime of registers to free more registers and reduce spill code.

Global Common Subexpression Elimination

Controls: toggle *Global_CSE*

The compiler eliminates common subexpressions across all basic blocks of a function. Also, any register computation that is invariant within a loop is moved above the loop.

Back-Substitution of Epilogue Code

Controls: toggle *Back_substitute_epilog*

Whenever there is an unconditional jump to the epilogue of a function, the compiler replaces the jump with the epilogue sequence. This optimization increases execution speed at the expense of code space.

Instruction Scheduling

Controls: toggle `Low_level_scheduling`

The compiler re-orders machine instructions to avoid any processor idle time; when an instruction waits for one of its operands to be computed by a prior instruction. Unrelated instructions are inserted in between two such instructions so that there will be (hopefully) no such idle time. In most architectures floating-point instructions take the most execution time (SCDE Release 2.01 or later).

Therefore floating-point instructions are started at the earliest possible time, and use of their results is delayed until the results become available.

Instruction scheduling takes extra compile time. Debugging fully scheduled code at the assembly level can be difficult, and scheduling also impairs symbolic debugging (option `-g`). We do not recommend using instruction scheduling in a debugging context.

Logical Predicates

Controls: toggle `NCRsetcc`

To reduce code size and branches, use the iX86 instruction `setcc` instead of an `if` expression to generate a logical 0 or 1.

Level 3 Optimizations

Level 3 includes all the optimizations performed at level 2, plus the following.

Instruction Scheduling

Controls: toggle `Low_level_scheduling`

The compiler re-orders machine instructions to avoid any processor idle time; when an instruction waits for one of its operands to be computed by a prior instruction. Unrelated instructions are inserted in between two such instructions so that there will be (hopefully) no such idle time. In most architectures floating-point instructions take the most execution time (SCDE Release 2.01 or later).

Therefore floating-point instructions are started at the earliest possible time, and use of their results is delayed until the results become available.

Instruction scheduling takes extra compile time. Debugging fully scheduled code at the assembly level can be difficult, and scheduling also impairs symbolic debugging (option `-g`). We do not recommend using instruction scheduling in a debugging context.

Peephole Optimizations

Controls: toggle `low_level_peep`

The compiler substitutes one code sequence for another that is more efficient. This may be process or specific, that is, depending on the target processor different substitutions will be made.

Transformations include:

- size reduction substitutions for constants and memory references
- specific instruction sequence substitutions for performance
- branch reduction substitutions
- dead code removal

Improved Spill Analysis

Controls: toggle *Cleanup_spills*

If the register allocator produces spill code, the compiler performs local common subexpression elimination again to eliminate any unnecessary spills.

Inlining

Controls: **-Hit=24**, **-Hic=24**

When a function or iterator (a High C extension) is inlined, calls to it are replaced with the logic contained in the function. This speeds up execution, but code size is potentially larger. The substitution saves prologue, epilogue, and parameter-passing overhead, and exposes opportunities for additional optimizations, such as copy propagation and constant folding.

You can select individual functions to be inlined, or allow automatic selection of functions according to criteria you provide.

For manual selection, use type qualifier `_Inline` to cause a particular function to be inlined (you must also specify **-Hi** on the command line). For example:

```
_Inline void f ( ) {
    . . .
}
```

For automatic selection, use the following command-line options to specify selection criteria. These options and others available for inlining are described in greater detail in Appendix D, “Function Inliner.”

- **-Hia**—Functions that have their address taken are not inlined.
- **-Hic=n**—Functions called less than **n** times are inline.

We provide a default value for **n** in the *hc.cnf* file. The default value is used only when the optimization level is greater than or equal to 4 and you do not specify a value for **n** on the command line.

- **-Hin**—Nested functions are not inlined. (a High C extension)
- **-Hir**—Recursive functions are not inlined.

- **-His=n**—Cause routines that use less than **n** bytes of local stack space to be inlined. The default value of **n** is unlimited. (High Performance C only).
The **-Hit**, **-Hic**, and **-His** options work together to qualify functions for inline expansion at function call sites. Any function in the source file is a candidate. Each criteria must be met for routine to be expanded inline. See the **-O** option for more information on when the inliner is invoked.
- **-Hit=n**—Functions with less than **n** tree nodes are inlined. There is roughly one tree node for each operator, operand, or other language construct such as a statement, loop, declaration, or function.
A default value for **n** is provided in the *hc.cnf* file. The default value is used only when the optimization level is greater than or equal to 4 and you do not specify a value for **n** on the command line.
- **-Hiu**—Mnemonic inliner-invented names are used for multi-module inlining (SCDE Release 2.01 or later).
- **-Hiw**—The inliner is invoked in multi-module mode (SCDE Release 2.01 or later).
- **-Hix**—Exported functions are not inlined.

Enhanced Frame Removal

The function allocates local storage space for variables on the stack, but will use an additional register to hold the address of the local area start. This is called the function's frame. It is set up at the start of a function and is destroyed at the end.

Enhanced frame removal uses the stack pointer to hold this address and simplifies the setup/destroy of the frame. For frequently called functions that are small, this will help performance.

This option should not be used when debugging with **sdb** (SCDE Release 2.01 or later).

Resolve Branch Targets

This is useful for **-486** compiles. The target of control transfer is aligned with pads to avoid cache pipeline stalls. These occur when an instruction spans two 16-byte cache lines (SCDE Release 2.01 or later).

Argument Passing in Registers

Controls: toggle *ABI_Args_in_regs* (enabled unless -kPIC)

This option passes the first two arguments in registers instead of on the stack to minimize call overhead. For global functions, this will generate an alternate entry point for internal function calls called *__AIR_name*, where name is the function name, since an extended call must be iABI compliant and pass its arguments on the stack.

Additional Low level Scheduling

Controls: toggle *Low_level_scheduling_pass2*

This option performs an additional pass after low level scheduling to better optimize the code.

New Call Frame

Controls: toggle *New_call_frame*

This option generates smaller function prologue and epilogue sequences to optimize function calling.

Frame Pointer

Controls: toggle *Use_frame_pointer*

This option, if Off, eliminates the use of **ebp** as a frame pointer.

Pointer-Based Variables

Controls: toggle *Behaved* (works at any level; default is On at level 4)

The compiler can be less conservative in generating code for pointer-based objects in a program that follows these rules:

- The address of a union member is never assigned to a pointer
- A value of a pointer type is never cast to an incompatible pointer type

With these assumptions, the compiler might be able to generate substantially faster code in referencing pointer-based variables.

Level 5 Optimizations

Level 5 includes all the optimizations performed at level 4, plus the following.

Loop Strength Reduction

Controls: toggle `Strength_reduce`, `induction-analysis`

In the body of a loop, multiplications and array-indexing operations involving the loop counter can be replaced by additions. This optimization usually speeds up operations that are performed on every iteration of the loop. However, it could actually slow down execution if rarely executed multiplications (as in a switch statement) are replaced by additions that are always executed (SCDE Release 2.01 or later).

Level 6 Optimizations

Level 6 includes all the optimizations performed at level 5, plus the following.

Loop Unrolling

Controls: pragma **Loop_factor**, option **-Hunroll**

Loop unrolling (or loop unwinding) involves replacing a loop with the code that makes up the loop body, replicated for some number of loop iterations. This optimization always increases code size. See Appendix G, “Loop Unrolling” (SCDE Release 2.01 or later).

Interprocedural Optimizations

Controls: **toggle** *interprocedural_optimizations*

Functions in the module being compiled will be processed in ‘callee-caller’ order, so that information useful to the ‘caller’ about the ‘callee’ function can allow additional optimizations to be done.

Level s Optimizations

Level s includes one space-saving optimization plus several time- and space-saving optimizations that also appear at other levels (SCDE Release 2.01 or later).

Optimizing for Space

Controls: toggle `Optimize_for_space`

Other Level s Optimizations

The following level s optimizations are also included at one or more numbered optimization levels:

- Preloading Arguments from Memory
- Cross Jumping (Tail Merging)
- Loop Memory-Reference Elimination
- Register Lifetime Analysis
- Spill Analysis
- Enhanced Register Allocation

Cache Optimizations

Similar to **-Os**, this level (**-Oc**) should be used for large programs that exceed the size of the processor and secondary cache. Efforts are made to reduce the cache footprint for large cache-intensive applications. It can have a negative performance effect on small applications that do not saturate the cache. A reduction in the object size should be seen with this feature. See the “Basic Block Optimizing” section in this appendix.

Processor-Specific Code Generation

The compiler has the ability to target code generation for the 80386, 80486, Pentium, or Pentium Pro processors. Several options are available for generating code that is either highly tuned for a specific processor or is "blended" to run on all of the processors.

Users who plan to have their executables run on several models of System 3000 systems should use the blended code strategy for generating their applications. This will insure acceptable performance and compatibility across all systems. But for specialized applications or benchmarks that are targeted for maximum performance on a specific system the code generation can be improved by specifying the processor specific options.

For example, the default code generation for SCDE 3.01 is code blended for Pentium Pro, Pentium, and 80486 based systems. The libraries and utilities that are part of SCDE 3.01 were compiled with this code generation strategy. This should give the best performance across all System 3000 systems. The -586 option produces blend code for the 80486 or Pentium processors, and the -686 option produces blend code for the 80486, Pentium, and Pentium Pro processors. The code, however, may not be the most optimal for Pentium Pro systems.

To get the best performance on a Pentium Pro based system, the compiler needs to generate code tuned specifically for this processor. The -**Hon=p6_only** option can be used in conjunction with -686 to specify code targeted to the Pentium Pro, but it may use instructions that are only supported on the Pentium Pro. Options for Pentium (-**Hon=pentium_only**), 80486 (-**Hon=486_only**), and 80386 (-**Hon=386_only**) specific code generation are also available.

Note that the code generated for a Pentium Pro is specific for the Pentium Pro only, and likely does not run on other processors. This is not true for Pentium, 80486, or 80386 specific code generation, although the performance will suffer on processors other than the target processor.

See "DEFAULTS FILE" in the **cc(1)** manual pages for information on how to change the default code generation strategy for a specific session or for all compiles on your system.

Basic Block Optimizing

The basic block optimizing feature provides the capability to optimize programs based on an actual dynamic execution trace, taken for sample runs through the program. First, the compiler instruments the program with code to generate the control flow trace. Next, the program is executed with a representative work load to generate the trace information. Finally, the trace information is used to re-optimize the program, targeting optimization to the most frequently executed paths.

A compile mode (**-qb**) adds code to programs to generate an execution trace of the program as it executes. Library functions (**-lbbarc**) process the trace and convert it into a representation of the program control transfer count data. The count data is output to a file (bbarc.out by default) when the program exits. The control flow count data file is then used to re-optimize the program, using the **-Hbbarc=filename** option to the compiler.

Utilities for Basic Block Optimization

The following utilities help with basic block optimizing:

- **dumpcnt(1)** or **profcnt(1)** is used for looking at the count data.
- **mergcnt(1)** is used for merging the count data for several runs.

Refer to the manual pages for instructions for running these utilities.

Types of Basic Block Optimizations

Optimizations performed include the following:

- reorganization of the program flow based on the flow graph counts to minimize the costs of branching
- reducing the program cache memory fetches
- cloning basic block in the highest count areas to minimize the cost of branching
- register assignment with priority given to the highest count paths

- special optimization of switches to shorten the time to the highest count areas

In addition, limited cloning of small block paths is done for high count paths if the toggle `-Hon=ncr_superblock` is enabled.

When to Perform Basic Block Optimization

Typically, basic block profiling and optimization should be used as a last optimization pass at the end of the development cycle, after debugging is complete. This is due to the extra effort required to gather the profile data and keep it up to date with the program. Any changes to the program structure that add new control paths (for example, if, for, while, or ? operator statements) invalidate the profile data set, since new program flow paths in the program no longer match the control flow paths represented in the count data file. Identical performance options and optimization levels must be used for the **-qb** compile and optimization compile using the count data generated by the **-qb** compiled executable.

When the **-Hbbarc** option is used, the SCDE 3.02 compiler invokes a new basic block ordering algorithm (*aggressive_cloning*) which clones certain small basic blocks that are frequently executed. The *aggressive_cloning* toggle reduces the cloning thresholds, thereby cloning more basic blocks and further reducing the number of "taken" branches. When this toggle is switched on, the size of the applications binary may considerably increase, thereby polluting the cache. Switch this toggle on only for small applications.

Options for Basic Block Optimization

The shell variable `BBPROFOPTS` can be set to a string of options to control the generation of the control data count file. The format for option string is a set of the following options, separated by commas:

- **dir**=directory_name—specifies where to build the profile count file.
- **path**=file_pathname—specifies the path name of the profile count file.
- **pid**=y—adds the process ID to the count file name.

- **merge=y**—merges new data with existing count file data.

The default profile data file name is *bbarc.out*, generated in the current working directory.

You should add process ID numbers to profile names if the program being profiled runs as several processes, so that a unique count file is generated for each process. To add process ID numbers, use the **pid=y** option. The default profile name is extended to *bbarcNNNNN.out*, where NNNNN is the process ID number.

For example, the following options generate the count files in the /tmp directory with a name that includes the process ID as /tmp/bbarc12345.out:

BBPROFOPTS="**dir=/tmp,pid=y**"

Using the **merge=y** option with an existing file name creates the file name.

Additional Ways to Optimize Performance

This section describes optimizations that are independent of the optimization level. You must explicitly request these optimizations using command-line options, pragmas, or toggles. Each description shows the controls you use to invoke the optimization.

Tail Recursion

Controls: toggle *Tail_recursion*

If a function calls itself recursively, and the call is the last action the function performs, the code can be modified to branch to the beginning of the function instead of calling it again, with appropriate adjustments for parameter passing and return. When you turn On toggle *Tail_recursion*, the compiler optimizes any recursively called function that can be modified this way.

Function Prologue/Epilogue Optimizations

Controls: toggles *Optimize_FP*, *Use_frame_pointer*, *486*

If a function does not store any local values, the compiler might be able to eliminate saving and restoring the `ebp` register when the function is called. Turning On toggle *Optimize_FP* makes this possible. For 486 machines, turning On toggle *486* and turning Off *Use_frame_pointer* may allow use of **esp** instead of **ebp** to access the stack frame, freeing up `ebp` for other uses.

Function Calling Conventions

Controls	pragma	Calling_convention
	toggles	<i>Args_in_regs_for_locals</i> <i>Callee_pops_when_possible</i> <i>Postpone_arg_pops</i>

Passing function arguments in registers instead of on the stack can make code both smaller and faster.

For the Intel 80486, up to eight bytes of arguments can be passed this way to a static function. (You can pass up to two arguments, the sum of which is no more than eight bytes). To achieve this effect, turn On toggle *486* and also toggle *Args_in_regs_for_locals*; or use pragma *Calling_convention* to specify the `ARGS_IN_REGS` attribute.

If a function meets certain criteria, the function can be made responsible for popping its arguments from the stack, eliminating a need for the caller to restore the stack after the call. Use toggle *Callee_pops_when_possible* for this effect. Alternatively, in some cases the compiler might be able to postpone popping arguments until another function is called; use toggle *Postpone_arg_pops* for this. These two toggles are mutually exclusive, and neither is effective in 486 mode (that is, when toggle *486* is On, or option -**486** is specified on the command line).

Constants in Code

Controls: toggle *Read_only_strings*

When toggle *Read_only_strings* is On, all string constants are assumed to be read-only. This permits the compiler to place them in read-only sections, thus reducing system overhead when more than one process is executing the same code and reducing paging during execution.

Note: String constants qualified with type **const** are unconditionally placed in the read-only data section.

Processor- Specific Instructions

Controls: options **-386**, **-486**, **-586**, **-686** toggle *p6-only*

If you can be sure your program will be used only on machines with a specific CPU (or an instruction-compatible CPU), you can direct the compiler to generate instructions specific to that processor, and gain an increase in speed.

Code compiled with toggle *486* turned on also runs on a 386, but with some performance degradation.

Aligning Code and Data

Controls: pragma **Align_Members**

toggles *Align_all_labels*, *Align_labels*, *Align_routines*

Execution speed can be improved when labels and function entry points are aligned on boundaries appropriate to the processor instruction size.

References to structure members are faster when the members are aligned on appropriate byte boundaries. Whether there is much of a space trade-off depends on how many structures your application contains and how they are organized.

Multi-Module Inlining

Controls: options **-Hiu**, **-Hiw**, **-Hwide**, **-Hiz**

You can specify option **-Hiw** to make the inliner operate on a group of files being compiled as a collection, rather than on each file individually. This is known as multi-module inlining or wide inlining. When you are using separate source files for data-hiding purposes or because more than one programmer is working on a project, inlining a function from one module into another can markedly improve performance.

Also, benchmark results may be much improved where a benchmark is split across multiple files.

To achieve multi-module inlining you must also specify **-Hwide**, for “wide” compilation. See Appendix D, “Inliner Function” for more information about multi-module inlining.

Expression Simplification

Controls: toggle *Use_reciprocal_for_divide*

A floating-point divide operation whose quotient is a constant can be converted to a multiplication by the reciprocal of the constant. Floating-point multiplication is significantly faster than division in most cases. However, if the reciprocal of the constant is not exactly representable, multiplication may introduce an inaccuracy in the last bit of the resulting mantissa. In most programs this does not matter, you must decide whether this level of accuracy is critical in your application. Toggle *Use_reciprocal_for_divide* must be turned On for this effect; and it works at all optimization levels.

Applying the law of associativity can also reduce the number or complexity of instructions required. For example, $k(x + k1) + K2$ becomes $x + (k1 + k2)$:

```
float x, y;
int k1, k2;

    . . .
    y = (x + k1) + k2;
Computing (x + k1) + k2 requires:
1. convert k1 to a float
2. add x and k1
3. convert k2 to a float
4. add intermediate result and k2.
Computing x + (k1 + k2) requires:
1. add k1 and k2
2. convert this result to a float
3. add this to x.
```

Compiler Intrinsics

Calls to several library functions are replaced by intrinsic code. The code generated for computer intrinsics is fast because the compiler generates object code for these functions directly, saving the overhead of the function call, and tailoring the code to the “arguments.” High C supports the following intrinsics:

```
_abs(x)
_find_char (*p, search_length, search_char)
_inb (port)
_ind (port)
_insb (port, buf, count)
_insd (port, buf, count)
_insw (port, buf, count)
_inw (port)
_max (x, y)
_min (x, 6)
_outb (port, data)
```

```
_outd (port, data)
_outsb (port, buf, count)
_outsd (port, buf, count)
_outsw (port, buf, count)
_outw (port, data)
_outw (port, data)
_scasb_eq (*buf, c, len)
_scasb_ne (*buf, c, len)
_scasd_eq (*buf, c, len)
_scasd_ne (*buf, c, len)
_scasw_eq (*buf, c, len)
_scasw_ne (*buf, c, len)
_stosb (*buf, c, len)
_stosd (*buf, c, len)
_stosw (*buf, c, len)
```

If you wish to call an equivalent library function instead of an intrinsic, you must provide an external declaration. For example:

```
extern void _abs( );
```

would disable the **abs** intrinsic.

Function Call Count Reporting

The call graph feature provides the ability to discover what function call activity occurs for each basic block. Command line option **-qb** includes this. The compiler annotates each basic block of the compiled program with a list of the functions called from that basic block. Only functions whose names are known at compile time are listed. Functions called through pointers are not listed.

A program that was compiled and linked with the **-qb** option writes an execution trace file when it exits normally. This execution trace file supports all of the functions of **profcnt(1)** utility for basic block execution reporting. In addition, there are two new **profcnt** options that produce reports about program structure reference and function call activity:

- The **-c** option tells **profcnt** to report only the function call data and omit the block execution counts. Each line of this output has this form:
count calling-function called-function

The lines are sorted by the counts. It is sometimes useful to pipe the **profcnt** output to the **cgsort(1)** utility. The unprocessed **profcnt** output may not always place all like function calling-called pair data on one output line.

The compiler supports using the call count information in two ways:

- The **-F=file** option tells the inliner to put the called function inline in the calling function no matter what the size of the called function. Normally, you would manually select the hot function calls that appear at the top of the list for site-specific inlining.
- The **-zcallgraph=file** option tells **ld(1)** to cluster the functions that call each other together into virtual pages in the executable. This reduces the number of instruction cache reads and page faults for the hot parts of the program.

The **-C** option tells **profcnt** to report function call data with source module data. Each line of this output has the form:

```
count calling_function [source file] called _function
[source file]
```

If you pipe the output of **profcnt** through **cgsort**, the output file is sorted by the count field. The **modules** program can be used to produce the control files for the **-F=file** and **-Fcallgraph=file** command line options.

See the **modules** manual page for details on using the **modules** program.

Structure Usage

The structure report feature provides the ability to discover what structure reference activity occurs for each basic block. Command line option **-qs** adds this feature to those supported by the basic block profiling (**-qb**) option. It annotates each basic block of the compiled program with a list of the structure members that the block accesses. It also notes in the list whether the value of the member was read, written, or both read and written.

If both the **-qb** and **-qs** options are used on a compiler command line, the last (rightmost) one overrides the first (leftmost) one.

A program that was compiled and linked with the **-qs** option writes an execution trace file when it exits normally. This execution trace file supports all of the functions of **profcnt(1)** utility for basic block execution reporting (just as it would if the program had been compiled with the **-qb** option). In addition, there are two new **profcnt** options that produce reports about program structure reference activity:

- The **-s** option tells **profcnt** to intersperse structure reference and function call lines in the report. For each basic block that called a function, a line like
12345 CALLED bozo

is in the output. The number is the number of calls of "bozo" that were made from the basic block.

For each structure that was referenced by the basic block, a line like this is in the output:

```
12345 structName { <a[4] >b[12] <> c [60] }
```

The example line above says that the block made 12345 references to structure "structName." Member "a" was read and not written. Member "b" was written and not read. Member "c" was both read and written. The displacement in the structure of member "a" is 4, the displacement in the structure of member "b" is 12, and the displacement of structure member "c" is 60. Structure usage information should be examined to see which structures that are frequently referenced should be reorganized to reduce the number of data cache lines that the program reads.

- The **-S** option tells **profcnt** to report only the structure access data and omit the block execution count and function call data. It is sometimes useful to pipe the **profcnt** output to the **susort(1)** utility to get the output sorted and collected by structure reference characteristics. The unprocessed **profcnt** output may not always place all like-structure reference data on one output line.

For a more complete treatment of these features, see Appendix H, “Structure Report and Call Graph Facility”.

Intermediate Compilation

The intermediate compile feature allows a user to achieve the effects of wide compilation without making extensive makefile changes. This is especially useful when the source files that are involved are distributed across more than one directory.

The **-ilobj** command line option requests that the compiler store intermediate code in object files (files with a **.o** suffix by default). The compiler normally produces **.o** files that contain the **.text** and **.data** sections for the corresponding source (**.c**) modules. A command line such as

```
cc -c -ilobj xxx.c yyy.c zzz.c
```

produces files **xxx.o**, **yyy.o**, and **zzz.o** that contain intermediate code. A subsequent command line such as

```
cc -o prog xxx.o yyy.o zzz.o
```

that refers to these intermediate files causes this sequence of events (orchestrated by **cc(1)**—the compiler's control module):

1. **cc** passes **ld(1)** the **-ilobj** option and the names of two files. **ld** puts the path names of the intermediate object files that it sees into one of the files. It puts the path names of the normal object files that it sees into the other file.
2. **cc** reads the file that lists the intermediate object files. It extracts all of the intermediate files into temporary files and presents them all to the inliner module of the compiler at once, just as though they had been compiled together in one wide compile. **cc** passes any inliner control options (for example, **-Hit=n**) that appear in the command line to the inliner when it is processing the set of intermediate object files. By using these control options along with the **-Hih** option, the user can observe and control the amount of cross-module inline substitution that occurs during the inlining step. It is also generally beneficial to use the **-Hiz** option along with the inliner control options to request cross-module passing of function arguments in registers during the inlining step.

As an alternative, the user can specify the call graph output from a **-qb** compilation with an **-F=file** command line option. See the previous sections “Structure Usage” and “Function Call Count Reporting” for more details.

3. The inliner produces the intermediate code for the optimizer for each of the modules whose intermediate code is passed to it by **cc**. **cc** directs the optimizer to compile each object file, and to put the normal object code that results in a temporary file whose name it gets from **cc**.
4. Finally, **cc** directs **ld** to combine all the temporary files and the normal object files that **ld** saw in its first pass into the final executable file.

For a more complete discussion of intermediate compilation, see Appendix I, “Intermediate Compilation”.

Inline Function

This appendix covers the following inline topics:

- What Is An Inline?
- Options
- Option Descriptions
- Detailed Description of Options

What Is An Inliner?

An inliner replaces calls to routines with the logic contained within the routines, thus avoiding the overhead of routine calls at the expense of potentially larger code size. The inliner back-substitutes routine bodies where the routines are called, with parameters appropriately replaced with local variables or constants. This saves at least the routine prologue, epilogue, and parameter-passing overhead, and exposes opportunities for additional optimizations, such as copy propagation and constant folding, especially when the arguments themselves are constants.

Operationally, the inliner sits between the compiler's front end and the common optimizing back end. You can specify particular routines to be inlined, and also allow the inliner to automatically select routines that satisfy adjustable criteria, such as the number of calls and the size of the routine. Use the type qualifier `_Inline`, as in `_Inline void f(){...}`, to cause `f()` to be inlined. You must use command-line option `-Hi` to invoke the inliner in non-automatic mode so that `f()` will be inlined.

Alternatively, you can specify options to turn on both manual and automatic inlining, and to set the designated options.

Options

Option Descriptions

The following list contains a brief description of the options. See the section “Detailed Description of Options” for more information.

- **-Hit=n**—Cause routines with less than **n** tree nodes to be inlined. (There is one tree node, roughly, for each operator, operand, or other language construct, such as a statement, loop, declaration, or routine.)
- **-Hic=n**—Cause routines called less than **n** times to be inlined.
- **-His=n**—Cause routines that use less than **n** bytes of local stack space to be inlined. The default value of <count> is unlimited. (High Performance C only).

The **-Hit**, **-Hic**, and **-His** options work together to qualify functions for inline expansion at function call sites. Any function in the source file is a candidate. Each criteria must be met for a routine to be expanded inline. See the **-O** option for more information on when the inliner is invoked.

- **-Hir**—Do not inline recursive routines.
- **-Hix**—Do not inline exported routines.
- **-His**—Do not inline non-prototype functions that are forward referenced.
- **-Hin**—Do not inline nested routines (nested routines are a High C extension).
- **-Hia**—Do not inline routines that have their address taken.
- **-Hih**—Print history of which routines are inlined or “cloned”. (Cloned means copied within a containing routine that is being inlined.)
- **-Hiz**—For functions in a wide compile that are not inlined, generate arguments in registers across modules.
- **-Hiw**—When specified with the **-Hwide** option, multi-module wide inlining is done if more than one source module is specified. See the **cc** manual pages for more information on “WIDE COMPILING” and “INLINING” (High Performance C only).

The options for determining inlinability are overridden if a routine is explicitly designated as inlined with the **`_Inline`** type qualifier. The **`-Hit`** and **`-Hic`** options work together. If both are specified—and are not zero, then both conditions must be met. Otherwise, if the specified condition is met, the routine is inlined.

Routines chosen for inlining that are exported, recursive, or have their address taken are inlined; and they are also defined. Routines chosen for inlining that are not exported, not recursive, and do not have their address taken are inlined; but the definition is discarded. Recursive routines are inlined only once and then a call is generated.

The optimization level also determines if the inliner is invoked for a compile. For optimization levels 0, 1, and 2, the inliner is not automatically invoked. It is invoked for levels 3 and above. Default values for the function size (**`Hit=24`**) and call count (**`Hic=24`**) that are safe for general compiles are used.

Note that the default optimization level is -O3 if none is specified, so the inliner runs as a default. To stop it from running, use the **`-Hit=0`** option.

Routines nested within inlined routines are, as a rule, inlined as well; if they must be defined, they may be cloned, which results in multiple copies of the same routine for each inlined instance of its parent. Routines containing calls to the intrinsic function `_alloca` are not inlined.

Detailed Description of Options

The following list contains a more thorough description of the options.

- **-Hit=n, -Hic=n**—These two options provide the basic mechanism for adjusting the criteria that determine whether a routine is inlined or not. The **-Hit** option designates a tree-node threshold while the **-Hic** option designates a call-count threshold. When the **-Hit** option is used—and is not zero, routines with less than *n* expression-tree nodes are automatically inlined. When the **-Hic** option is used—and is not zero, routines that are called less than *n* times by direct calls are automatically inlined. If both options are used—and neither is zero, they work together; that is, both conditions must be met before a routine will automatically be inlined. That a routine is inlined does not mean that it will not also be defined; other factors such as recursivity, addressability, and exportedness play roles in determining whether the routine must be defined. If the routine can be completely inlined, it is not defined.
- **-Hir**—If a recursive routine (either directly or indirectly recursive) meets the criteria for inlining, it is inlined only at the first level and then a definition is generated so that it can be called at subsequent levels. It might be desirable to completely avoid inlining of recursive routines. The **-Hir** option suppresses inlining of recursive routines, even at the first invocation level. See the definition of a recursive routine below for the **-Hih** option.
- **-Hin**—Routines that are statically nested within inlined routines are usually inlined as well. This can be suppressed by designating the **-Hin** option. However, it should be noted that, in many cases, this causes nested routines to be cloned, resulting in a new instance of each nested routine for each inline copy of the parent.
- **-Hix**—Exported routines may be inlinable, but still must be defined in order to be exported. The **-Hix** option suppresses inlining of exported routines.
- **-Hia**—Routines that have their address taken may be inlinable, but still must be defined so that they have an address. The **-Hia** option suppresses inlining of such routines.

- **-Hih**—The **-Hih** option prints to standard output a history detailing which routines were defined, which were inlined, and which were cloned. The static nesting level and parent ID (a compiler-assigned number) are printed along with the ID and name of the routine, the size in tree nodes, and the number of calls. Following each entry is a list of status flags for the routine. The status flags are as follows:

```

DEF  Routine must be defined.
INL  Routine may be inlined.
CLO  This is a clone of a previously
      defined routine.
UPL  Routine makes up-level references.
REC  Routine is recursive.
ADR  Routine's address is taken.
ALO  Routine contains call to alloca().

```

For the purposes of **-Hih**, a recursive routine R is one for which inlining is requested, and is such that a call to R is produced in the process of inlining R. A routine for which inlining is not requested is not regarded as recursive. If a routine F calls G and G calls F, but only F is requested to be inlined, neither is marked as recursive because the inlining of F does not produce a call to F; it produces only a call to G.

If a routine is a clone, the ID of the original routine is given in parentheses immediately after the new ID number. Entries are indented to indicate their nesting/inline level.

Sample output:

```

= = > Proc2() {size=23, stack=0x8, calls=1} inlined, defined
      because is exported

```

A clone routine is a copy of a non-inlined routine F whose parent was inlined. There must be a clone for each distinct static nesting level in which F must be defined. Note that clones show up only if High C nested routines are used.

Other Constructs and Utilities

This appendix covers the following topics:

- Intel 80486 Versus 80386
- Intel Pentium Versus 80486
- Intel Pentium Versus 80486
- Preprocessor Predicates
- The Enhanced ASM Facility

Intel 80486 Versus 80386

The Intel 80486 is 100% binary compatible with 80386, but re-engineered for higher performance. Many of the 80486 instructions run in a single clock cycle.

When the **-486** option is specified, the compiler assumes that it is generating code that runs primarily on a 80486. It, therefore, generates alternate code sequences that run faster on a 80486 than a 80386, perhaps at the expense of code space.

For example, the `push <ea>` instruction (push from memory) runs slower on an 80486 than the two-instruction sequence:

```
mov     eax, <ea>
push    eax
```

Therefore, the latter is generated when **-486** is specified.

Because the `mov` instruction typically runs in one clock cycle on the 80486 (assuming a cache hit), performance can be improved by avoiding the `push` instruction altogether when passing arguments. Instead of pushing arguments in reverse on the stack, a fixed-size argument-list build area is allocated in a routine's prologue that is large enough to contain the largest argument list to be built within the routine. This build area is placed at the end of the stack frame so that its address is contained in **esp**. Arguments are stored into the build area from left to right at consecutive offsets, subject to boundary alignment. This scheme obviates the need to `pop` the stack after the call.

Intel Pentium Versus 80486

As with the 80486 and 80386, the Pentium is 100% binary compatible. When **-586** is invoked, all of the **-486** optimizations occur with additional code scheduling and selection for most optimal Pentium performance.

Using **-586** (the default for SCDE 2.02 and later) generates close to the best code for 80486 processors. Some minimal improvement will be seen using **-486**. All libraries in SCDE 2.02 were generated with **-586**.

Intel Pentium Pro Versus 80486

The Pentium Pro is upwardly compatible with the 80386, 80486, and Pentium processors. It additionally introduces several new instructions that can be used to improve program performance.

Using **-686** (the default for SCDE 3.01 and later) generates code that runs on all processors and is tuned to balance each. Using **-686** with **-Hon=P6_only** generates code using the new Pentium Pro instructions and should run faster on a Pentium Pro, but may fail on the earlier processors.

Preprocessor Predicates

A predicate is a preprocessor construct enhancement in UNIX SVR4. Its purpose is to provide a name that can be referenced in a `#if` statement without affecting the name space of macros.

A predicate is defined by associating a token sequence to it with the `#assert` statement:

```
#assert predicate(token_sequence)
```

To test for an assertion in an `#if` statement, the following syntax is used:

```
#if #predicate(token_sequence)
```

The `#unassert` statement removes assertion. To remove all assertions against a predicate, the following syntax is used:

```
#unassert predicate
```

To remove a single assertion against a predicate, the following syntax is used:

```
#unassert predicate(token_sequence)
```

The following assertions are predefined:

```
#assert system(unix)
#assert cpu(i386)
#assert machine(i386)
#assert machine(_nst)
```

Note that the command line option `-A` can be used to preassert additional predicates. See the “Options” section in Appendix D, “Inliner Function” for more information.

The Enhanced ASM Facility

High C supports the enhanced ASM facility as defined in Appendix A of the *UNIX SYSTEM V Programmer's Guide: ANSI C and Programming Support Tools*.

Example:

```
asm char in(int port) {
%con port;
    in port
    %mem port;
    movl port,%edx
    in (%edx)
}
```

The compiler expects the asm macros to preserve non scratch registers, as a call would.

When writing asm macros, beware that argument loads like:

```
movl arg1, %oesi
```

may cause problems if `%oesi` is used in a subsequent argument reference. Only the scratch registers are sure to not be used in address nodes.

High C Language Extensions

- Introduction to High C Language Extensions
- X3J11 Extensions to C
- Multi-Character Constants
- String Replacement (#) and Token-Pasting (##) in Macros
- Trigraphs
- High C Extensions Documented in the Manual
- Named Parameter Association
- Nested Functions and Full-Function Variables
- Communication With Other Languages
- Intrinsics
- Packed and Unpacked Structures
- __HighC__ Predefined Macro
- Calling-Convention Attribute Specified with Function Declaration
- Shiftjis Kanji Support
- Direct I/O to Hardware Ports
- In-Line Placement of Bytes of Code
- ASM and Direct Register Assignment for Variables
- short/long enum
- ANSI Preprocessor Numbers
- Iterators
- Prototypes

Introduction to High C Language Extensions

This appendix presents features in the High C language not typically found in implementations of C.

For more information about these extensions, see the “C Language” section of the *UNIX SYSTEM V Programmer’s Guide: ANSI C and Programming Support Tools*.

Note: While these extensions provide valuable development aids, programmers are reminded that they may be High C exclusive. Code written using these extensions may be non-portable to other NCR and non-NCR systems, including future releases.

X3J11 Extensions to C

Using \ as a Line Continuation Character

To overcome source line limitations, any line ending in the character \ is treated as contiguous with the following line; therefore, any C word must be broken across line boundaries. The \ is most useful for multiple-line macro definitions and long strings, although there is a better solution for the latter (see the next item).

String-Constant Concatenation

Juxtaposed string constants denote the concatenation of the constants.

Example: The string

```
char *p = "Hi" "there,"  
         "folks.";
```

is equivalent to

```
char *p = "Hi there, folks.";
```

This feature is useful for long string constants that must span more than a line of text, and for aligning portions of a string to emphasize or illuminate correspondences.

Vertical Tab as a Delimiter

The ASCII vertical tab character is semantically equivalent to a blank when not appearing within a string or character.

Suffixes u(U) and l(L) in Integer Constants

These suffixes, either alone or together, in either order and independently in either upper or lower case, are permitted in integer constants: decimal, octal, and hexadecimal. The u changes the type T that the integer constant would otherwise have to the unsigned variety of T. The l changes T to the long variety of T. For example,

- 123 has type Signed-Int
- 123u Unsigned-Int
- “1234” Signed-Long-Int
- 123ul and 123lu Unsigned-Long-Int

Suffixes f(F) and l(L) in Floating-Point Constants

The suffix l, in upper or lower case, is permitted in floating-point constants. It makes the type of the constant Long-Double instead of the default Double. The suffix f, in upper or lower case, makes the constant’s type Float rather than Double. f and l cannot appear together.

Escape Sequences \a, \v, and \xddd in Strings and Characters

In addition to the escape sequences \n, \t, \b, \r, \f, \\\, \', \“ and \ddd (octal digits) allowed by KR, the sequences \a, \v, and \xddd are allowed, where

- \a denotes audible alert or the ASCII BEL character
- \v denotes the ASCII vertical tab character
- ddd is a sequence of one to three hexadecimal digits (with each letter in upper or lower case): \xddd denotes a single byte whose value is ddd.

New Reserved Word **signed**

The type modifier **signed** indicates the modified type is to be signed. This type modifier is most useful in guaranteeing that a **char** type is signed, since an implementation is free to decide whether the unadorned type denotation **char** denotes **Signed-Char** or **Unsigned-Char**. For example, **signed char c;** declares **c** of type **Signed-Char**, whereas **char c;** may be signed or unsigned, depending upon the implementation.

New type Denotation **long double**

The type specifier **long double** denotes a new floating-point type **Long-Double**, having precision and range no less than type **Double**. Use this denotation to obtain a triple-precision floating-point type.

Initialization of Automatic Aggregates With Static Expressions

An aggregate object **O** with storage class **automatic** may be initialized with the same initializing expressions permitted for static aggregates and additionally, if **O** is a structure or union, with a single expression of **O**'s type.

Initialization of Structures With Structure-Valued Expressions

A structure (**struct/enum**) may be initialized with a structure-valued expression, as in:

```
extern struct s{int x;} f();
main() {
    struct s S = f();
}
```

Function Prototypes

See “Brief Tutorial on Prototypes” for an introduction to this Pascal-like concept in C.

Multi-Character Constants

High C permits a character constant that contains more than one character. The value of a two-character constant is:

$$256 * (\text{value of first character}) + (\text{value of second character})$$

For example, the value of 'ab' is $256 * 'a' + 'b'$. In general, the value of an n-character constant is:

$$256 * (\text{value of first } n-1 \text{ character}) + (\text{value of last character})$$

String Replacement (#) and Token-Pasting (##) in Macros

The Standard operators # and ## are permitted within the replacement token sequence *R* of a definition of a parameterized macro. Within *R*, these operators have special meanings. These meanings change the way token substitution is performed for parameter names that appear as arguments to # and ##.

Normally, when a parameter name *N* appears in *R*, *N* is replaced by the token sequence that was given as the argument for *N* at the macro invocation, after macro expansion has first been performed on that token sequence. That is, macro expansion is first applied to the argument token sequence, then substitution of the expanded text occurs wherever *N* appears in *R*.

If # is followed by a macro parameter name *N* within *R*, both are replaced by a single character-string token that contains the spelling of the unexpanded argument token sequence for *N*; that is, before macro expansion has been applied to the argument. The spelling is defined as the concatenation of the texts of each of the individual tokens, with a single space inserted between any two tokens that have any amount of whitespace (comments, spaces, tabs, and so forth) between them, and where any occurrence of “ or \ is prefixed by \ (to ensure that when compiled the string produces “ or \ correctly).

If a macro parameter name *N* is immediately preceded or followed by ##, the parameter is replaced by the unexpanded argument token sequence. Before the replacement text is re-examined for further macro expansion, each instance of ## is deleted, and the two tokens preceding and following the ## are replaced by a single token whose text contains the concatenation of the texts of the two tokens. The resulting concatenation must be a valid token, otherwise the result is undefined. If there are no such two tokens, the result is undefined. The resulting single token is considered for expansion when the replacement text is re-examined for further macro expansion.

It is an error for ## to appear as the first or last token of a replacement token sequence.

A typical use of # would be for printing the text of a variable or expression in a program:

```
#define str(x)  #x
#define debug(x) \
    printf("The value of " str(x) " is %d", x)
```

If the int variable xyz has the value 3, the invocation `debug(xyz)` prints:

The value of xyz is 3

and `debug(xyz+ /*comment*/ 4)` prints:

The value of xyz+ 4 is 7

Notice the single space inserted between + and 4, corresponding to the whitespace between + and 4 in the argument to `debug`.

```
debug('a')
```

prints:

The value of 'a' is 97

(assuming an ASCII target) because 'a' is stringized into the C string 'a'.

A typical use of ## is for the construction of an identifier from a distinct identifier prefix and suffix:

```
#define Glue(x,y) x ## y
#define OS_CPU(Prefix) char *Glue(Prefix,_OS), \
    *Glue(Prefix,_CPU)
```

Then:

```
OS_CPU(VAX)
OS_CPU(SUN)
```

declares four char * variables: `VAX_OS`, `VAX_CPU`, `SUN_OS`, and `SUN_CPU` (assuming `VAX` and `SUN` are not parameterless macros that expand to something else).

To illustrate the substitution of non-expanded argument token sequences, consider the following:

```
#define Str(x)  #x
#define Str2(x) Str(x)
    #define abc def
```

Then:

```
Str(abc)    // expands to "abc"
Str2(abc)   // expands to "def"
```

First `abc` is expanded to `def`, then `def` is substituted in the replacement text for `Str2`, producing `Str2(def)`; then `def` is stringized.

For example:

```
#define Glue(x,y) x ## y
#define Glue2(x,y) Glue(x,y)
```

Then:

```
Glue(abc,abc)    // expands to abcabc.
Glue2(abc,abc)   // expands to defdef.
```

Finally, some error situations:

```
#define Bad1(x) ## x ##    // Both ##s are invalid.
#define Bad2(x) #y         // y is not a
    #define Glue(x,y) x ## y // parameter.
    Glue(,123)             // Invalid: there are no tokens
                          // to the left of ##.
```

Trigraphs

Trigraphs are intended for use in environments lacking the Standard C character set. Before any source-code lines are spliced, the replacements shown in the table below occur in source text anywhere a trigraph is found, including in strings and character constants:

Table F-1. Source Text Replacements

This expression. . .	Is replaced by. . .
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	
??>	}
??-	~

Note that the ANSI Standard has quietly changed the semantics of some old C programs. For example:

```
main () {printf("Sales are up (are you sure??) \n");}
```

now prints:

```
Sales are up (are you sure]
```


High C Extensions Documented in the Manual

Underscores in numbers

Numbers, both floating-point and integer constants, can be written with the character ‘_’ among the digits. Generally, ‘_’ takes the place of the English comma in numbers. For example:

```
1_000_000      /* One million. */
```

All characters in identifiers are significant

Mix of declarations and statements

In a compound statement, that is, the list of declarations and statements enclosed within { and }, declarations and statements can be mixed, as opposed to declarations preceding statements. This allows you to place a declaration near its first use, where its initial value may be available. For example:

```
int A(10);
Compute_array (A);
    int M = Max(A, 10);
```

With this relaxed order, it is now possible to execute (formally elaborate, a term borrowed from Ada) a declaration more than once. Elaboration includes initialization. For example:

```
Loop: ;
int K, Count = 1;
for (K = 1; K <= 10; k++) {
    ...Do_something(); ...
}
goto Loop;
```

Each time around the Loop, Count is initialized to one.

Case Ranges in the case Statement

High C permits the extension case E1.. E2: where the meaning is equivalent to case E1: case E1+1; case E1+2: ... case E2:. For example:

```
switch (Ch) {
case 'A' .. 'Z':      Scan_id();      break;
case '0' .. '9':      Scan_number();  break;

default:              Scan_delimiter(): break;
} /* The latter break for safety: */
  /* in case of reordering cases. */
```

Initialization of automatic aggregates with arbitrary expressions

High C permits automatic aggregates to be initialized with arbitrary non-constant expressions, where X3J11 restricts initialization lists to constant expressions.

New Reserved Word pragma

In High C, pragma is a reserved word. You cannot use the word pragma as an identifier. Pragmas are used to direct the compiler when more than one possibility is provided for an implementation-defined aspect of compilation. Subsequent to High C's choice of pragma, the ANSI committee chose #pragma for the same purpose. High C supports #pragma and continues to support pragma for compatibility. We suggest using #pragma for portability. However, ANSI'S #pragma cannot appear within macros, whereas High C's pragma can, so there is a definite advantage to pragma.

C++ Comment Style

High C supports the C++ form of comments: The character sequence // introduces a comment that extends to the end of the line. For example:

```
int i;    // This is a comment.
```

The disadvantage of introducing this form of comment is that the following Standard C program fragment, whose style is questionable, is no longer recognized as valid:

```
i = a /**divide a by: */ b;
```

When you compile in ANSI mode, however, the C++ comment is not allowed and the above fragment is correctly recognized.

C++ for Statement

In C++, the first expression and semicolon within the parentheses of a for loop can be replaced with a declaration or statement. High C allows this as an extension. For example:

```
for (int i = 0; i < 10; i++) . . .
```

declares `i` and initializes to 0 at the beginning of the loop. This is equivalent to:

```
int i = 0;
for ( ; i < 10; i++) . . .
```

which may still not be valid in Standard C if the declaration follows other statements, but it is valid in High C.

It is unfortunate that C++ did not restrict the scope of any declared names to the for loop. The semantics should have been defined in terms of:

```
{
int i = 0
  for ( ; i < 10; i++) . . .
}
```

however, the High C compiler adheres to C++ design for the sake of those already familiar with C++.

Named Parameter Association

Functions declared with parameter names can be called with the named parameter association syntax of Ada. Because such calls refer to the parameter names rather than their positions, the order of supplied parameters is irrelevant. The syntax is like that of a normal function call except that each actual parameter expression is preceded by the corresponding formal parameter name followed by “= >”. For example:

```
typedef enum {Red,Green,Blue} Color;
void P (int A, float B, Color C, Color D) { ... }
...
P (C = > Red, D = > Blue, B = > x*10.0, A = > y);
```

You can also start the function call using positional parameter notation and switch to named association as the parameters are written down from left to right. Switching back to positional notation is not allowed, nor is any other variation. For example:

```
void Plot (Xlo,Xhi,Ylo,Yhi,Xinc,Yinc)
float Xlo,Xhi,Ylo,Yhi,Xinc,Yinc; {
    ...
}
...
Plot (Alo,Ahi,Blo*2.0,Bhi*2.0,Yinc=>y,Xinc=>f(x+z));
```

Syntax

The syntax for arguments to a function call is expanded from:

zero or more assignment-expressions
separated by commas

which is the normal “positional” form for Standard C, to also include:

zero or more assignment-expressions
separated by commas
followed by
one or more identifiers
followed by
=> and an assignment-expression
separated by commas.

The effect is that there can be an initial portion of the argument list in the positional form (a “positional prefix”), followed by named-parameter association form (“named parameter association suffix”).

Thus

```
f(1, 2, 3, aa => x+y, bb => q);
```

is allowed, but

```
f(1, 2, 3, aa => x+y, bb => q, 7, 8, 9);
```

is not.

Constraints

Any function called with the named parameter association must specify the names of all of its arguments. For example, `int f(float g);` can be called this way, but `int f(float);` or `int f();` cannot.

The identifiers in a named parameter association form of call must collectively be the names of those parameters to the called function whose values are not supplied positionally. Values for all parameters must be supplied through either the positional prefix or through the named parameter association suffix.

Given these constraints, it is possible to transform the function call into a purely positional form. Then constraints and semantics are as if the arguments to the call were thus transformed. For example, the call to `P` in the first example in this section can be transformed from :

```
P (C = > Red, D = > Blue, B = > x*10.0, A = > y);
```

to:

```
P (y, x*10.0, Red, Blue);
```

and then the regular constraints and semantics of Standard C must be obeyed for the transformed call.

Nested Functions and Full-Function Variables

Nested Functions

In High C, functions can be defined within functions. Such functions are called nested. This facility endows High C with an expressive power that is found in Pascal.

In the body of a nested function *N*, any name in a containing scope may be used. That is, the body of *N* may use names local to other functions that contain *N*. This is called up-level referencing and any such names are said to be up-level referenced from *N*. The single restriction is that register-class variables cannot be up-level referenced.

Up-level-referencing can be achieved by making available to *N*, at each call to it, a way to reference the collection of locals of each of its enclosing functions. These collections are together called *N*'s environment.

The function *P* immediately enclosing *N* is called *N*'s parent; the next enclosing function *G* is its grandparent, and so on. The collection of locals of each function is called its stack frame.

(Technical Note: *N*'s environment may be implemented by passing to *N*, at each call to it, a hidden parameter that is a reference to *P*'s stack frame. If this is done for all functions, *P* has in its stack frame a similar hidden parameter that links it to *G*, and so on, out to the global level where functions need no such link. This is called the static link method of implementing up-level referencing, and it is the method of choice for best efficiency and optimization possibilities, as opposed to the display method, which is not described here.)

Therefore, a major difference between nested functions and non-nested functions is that the address of a nested function *N* does not entirely capture *N*'s "value": the environment is also required. In contrast, the address of a non-nested or global function *G* entirely captures its value, since *G* has no parent and thus needs no environment. The C notion of "pointer to function" is therefore sufficient to capture the value of *G* but not that of *N*.

Hence High C disallows taking the address of a nested function, and, where C assumes & before any expression of type function, High C does not assume the & if the expression is of a nested-function type.

Full-function Values

A function address and its environment is referred to as a full-function value, as opposed to just a function address.

All of the capabilities associated with global functions, such as passing them as parameters and storing their value into variables, are available for nested functions, although new syntax is required.

A variable capable of holding a full-function value, and therefore the value of a nested function, is declared as a function declaration would be, except that “!” follows the parenthesized formal parameter list. For example:

```
int ffv(!);
```

In contrast, a Standard C variable capable of holding only a function address is declared using the pointer syntax:

```
int (*fa) ();
```

`ffv` may be called with the expression “`ffv( )`”, but not with “`( *ffv )`”, since `ffv` is not (just) a pointer.

Any nested function may be assigned to `ffv`. A global function `G` may be assigned to `ffv` by dereferencing it, since of course `G` is transformed to `&G` by the compiler and must be dereferenced to obtain the full-function value of `G`, not just its address: “`ffv = *G`”. The environment stored in `ffv` in such an assignment is meaningless, since `G` needs no environment. Upon calling the value in `ffv`, the environment is passed to `G` but `G` (indeed every global function) safely ignores it.

Nested functions may be passed as parameters: the full value is passed. The full-function value of a global function may be passed by dereferencing the global function: the passed environment is meaningless.

An argument can be declared as being a full-function value by using the new syntax:

```
int f(ffv) int ffv(!); ( ... ffv(); ...)
```

Only the names of function constants may be dereferenced to produce full-function values. The dereference of a pointer to a function is immediately converted back to an address by Standard C rules for function expression conversion. Thus:

```
extern int sub();
main () {
    int (*fa) () ;
    int Nested () (...)
    main(*main);          /* Passes the full value of main.    */
    main(*sub);           /* Passes the full value of sub.    */
    main(*fa);            /* Passes fa, since *fa=>&*fa = fa */
    main(Nested);
    /*Passes the full value of Nested.    */
}
```

This extension is compatible with ANSI Standard C since the dereference of an expression of type pointer-to-function is permitted only in the context of an expression denoting a function to be called, for example, “(*fa)()”; but “(*fa)(*fa)” is illegal in ANSI C.

In the following example of full-function values, a call to a sort function takes as parameters two functions:

```
extern void Quick_sort (
int Lo, int Hi, int Compare (int a,int b),
    void Swap (int a,int b)
);
static Sort_private-table () {
Entry Entries [100];
    int Compare (int a,int b) {
        return Entries [a] < Entries[b];
    }
    void Swap(int a,int b){
        Entry Temp = Entries [a];
        Entries [a] = Entries[b]; Entries [b] = Temp;
    }
    ...
    Quick_sort(1,100,Compare,Swap);
}
```


Here it is necessary for `Compare` and `Swap` to be local to `Sort_private_table` since the table `Entries` is local to that function. In Standard C, `Entries`, `Compare`, and `Swap` would have been moved outside of `Sort_private_table`. This works fine in this simple case, but if `Sort_private_table` were recursive, one would have to explicitly manage a stack of `Entries` arrays to get the desired effect.

Although this example may seem contrived, a stripped down version of a practical Pascal program, translated into High C, is included in distributions of High C compilers. The name of the file is *analyze.c* and it implements a graph traversal algorithm. Any C programmer who thinks Standard C function capabilities are adequate should read this program and attempt to translate it to Standard C.

Casting any full-function value of one type to any full-function value of another type is permitted, in concert with the ability to cast function addresses in Standard C. The `sizeof` of a full-function type may be taken and is always greater than the `sizeof` of a function address, since the former includes the environment.

The additional syntax required to permit the declaration of full-function types follows, and is simply a repeat of the rules for Standard function syntax except that “!” is allowed.

Non-Local Labels

In High C, a function label is visible to the function in which it is declared and to all of the descendants of that function. A label declared in a descendant overrides the declaration of a label in an ancestor.

Jumping to an ancestor’s label is a valuable technique for easily unwinding a computation deep in the throes of recursion. It is a disciplined form of C’s `setjmp` / `longjmp` and comes from Pascal. For example:

```
void Wind (void Callme(!)) {
void P (int Cnt) {
    if (!Cnt) Callme();
    P(CNT-1);
}
P(100);
}
void main () {
```

```

void Called () {
    printf("%s%s", "Called from deep within"
    "recursion -- exiting. \n");
    goto EXIT'
}
Wind(Called);
printf("I should not get here.\n"); return;
EXIT:
printf("%s%s", "Deep recursion unwound by"
"a single goto.\n");
}

```

Here `main` invokes a procedure `Wind` that causes local function `P` to make 100 invocations of itself before calling function `Called`. `Called` disposes of all the invocations of `P` in one fell swoop by jumping to the `EXIT` label. The High C compiler emits the proper code to clean up the stack. In locally optimizing High C compilers, the operation is extremely cheap - just a few instructions - but no variables in the function containing the `EXIT` label can be allocated to a register. In globally optimizing High C compilers, each unwound stack frame must be visited individually to do the unwinding so that register variables can be used in the main function.

Although this example may seem frivolous there are in fact serious uses of non-local `gotos` in the High C compiler itself.

Communication With Other Languages

A High C module can communicate with other modules written in other languages partially by virtue of the pragmas **Calling_convention**, **Data**, and **Code**. Although the syntax of these pragmas is machine independent, their effects are sometimes machine dependent, and hence their documentation can be found in “Externals.”

Intrinsics

High C contains a set of so-called “intrinsic functions” that provide the ability to do the following:

- Take the absolute value, minimum, and maximum of values of any arithmetic type
- Move and compare bytes of memory using the host machine’s most efficient instructions.

Intrinsic functions need not be declared to be used.

Descriptions

Below is a list of the intrinsic functions followed by their descriptions in the same order as the list.

<code>_abs</code>	<code>_max/_min</code>
<code>_find_char</code>	<code>_skip_char</code>
<code>_move</code>	<code>_move_right</code>
<code>_fill_char</code>	<code>_compare</code>
<code>_offsetof</code>	<code>_usually/_rarely</code>

`_abs(x)`

x must be of an arithmetic type. The result is the absolute value of **x** and of the same type as the type of **x**.

`_max(e1,e2,...)/_min(e1,e2,...)`

e1,e2,... must be of arithmetic types. The result is the maximum/minimum of **e1,e2,...** .

More precisely, `_max(e1, e2)` is of type `T=Common` (type of **e1**,type of **e2**) and is the maximum value of **e1** and **e2**, each cast to type **T**.

`_max(e1, e2, . . . , en)`, where $n \geq 3$, is (recursively) defined to be `_max(_max(e1, e2), e3, . . . , en)`. The specification is analogous for `_min`.

For example:

```
float f; unsigned long ul; int i;
```

```
main () {
    _min(f, ul, i); /* Has type Float. */
    _min(ul, i);    /* Has type Unsigned-Long-Int. */
    -min(i, f);     /* Has type Float. */
}
```

It is guaranteed that the operands of `_max` and `_min` are evaluated, at most, once, unlike the Standard macro definition of `max (min): “#define max(x, y) ((x)>(y)?(x):(y))”`, where one argument is evaluated once and the other twice.

unsigned _find_char(any_pointer p, unsigned search_length, char search_char)

Searches `((char*)p)[0]` through `((char*)p)[search_length-1]` for `search_char` using the most efficient host instruction to do so. It returns the index `i` (a number in the range `0 ... search_length-1`) if it found the character, where `p[i] = search_char`; otherwise it returns `search_length`, indicating a failed search.

The Standard function `strlen` can be implemented using `_find_char`:

```
strlen(s) _find_char(s, 65_535, 0)
```

assuming at most 65,535 characters in a string. This version of `strlen` generates in-line code.

unsigned _skip_char (any_pointer p, unsigned search_length, char search_char)

Does the same as `_find_char`, except that it searches for the first character not equal to `search_char`.

void _fill_char (any_pointer p, unsigned len, char fill)

Fills memory from `((char*)p)[0]` to `((char*)p)[len-1]` with the fill character. This can be used to implement the Standard library function `memset`:

```
#define memset(p, fill, len) _fill_char(p, len, fill)
```

void _move (any_pointer from, any_pointer to, unsigned len)

Moves len bytes from the address ((char*) from) to the address ((char*)to). The move occurs from left-to-right (lower-to-higher addresses). If from < to < from+len, use _move_right below.

This can be used to implement the Standard library function memcpy:

```
#define memcpy(dest,src,len) _move(src,dest,len)
```

void _move_right (any_pointer from, any_pointer to, unsigned len)

Does the same as _move except the move occurs from right-to-left (higher-to-lower addresses). If to < from < to+len, use _move above.

int _compare (any_pointer p1, any_pointer p2, unsigned len)

Compares the string ((char*)p1) [0]...((char*) p1)[len-1] with the string ((char*)p2) [0]..((char*)p2)[len-1]. It returns 0 if they are identical, -1 if the first byte at which the two disagree is less in the first string than the corresponding byte in the other stream, and +1 otherwise.

**_offsetof(any_struct_type, any_field_name),
_offsetof(any_expression_of_a_struct_type, any_field_name)**

The purpose of _offsetof is to obtain the byte offset of a field in a struct or union (structure) type. _offsetof is actually an addition to the phrase-structure syntax rather than being a predefined function.

The syntax of the first argument is identical to that of sizeof; it is either an expression of some structure type or an Abstract_declarator of some structure type. The result has the same type as that of sizeof and is the byte offset from the beginning of the structure for the named field. The named field must, of course, be a member of the given structure type. For example:

```
struct s {char x,y; int i:3,j:2, int a[10]; } x;  
_offsetof(struct s, x) /* is 0 */
```

```

        _offsetof(struct s, y)      /* is 1 */
        _offsetof(x      , j)      /* is 2 on an 8086 */
*/
        _offsetof(x      , a)      /* is 3 on an 8086 (no padding)
*/

```

`_offsetof` is supplied to implement the `offsetof` macro required by the ANSI Standard for C:

```
#define offsetof(Type, Field) _offsetof(Type, Field)
```

However, `_offsetof` is more flexible in that it allows an expression as the first argument (like `sizeof`) instead of just a type.

`_USUALLY(conditional expression)` and `_RARELY(conditional expression)`

These intrinsics may be used during conditional branch statements to direct the compiler to lay out the code such that the most likely code path is the fall-through case. Note that their use is restricted to conditional jumps with two successors. They cannot be used with indexed jumps such as those generated by calls such as `(*func)()`, or with switch statements. When `_USUALLY(conditional expression)` is used, the compiler lays out the successor blocks such that the "true" case comes immediately after the conditional expression. When `_RARELY(conditional expression)` is used, the "false" case comes immediately after the conditional expression. Take as an example the following code:

```

#ifndef _USUALLY()
#define _USUALLY(expr) (expr)
#endif
#ifndef _RARELY()
#define _RARELY(expr) (expr)
#endif

main() (
    {int i;
      for (i= -2; i<4; i++)
        printf("func(%d) = %d\n", i, func(i));
    }
func(int a) {

    int ret_val;

    if (_RARELY(a < 0))
        printf("got a negative value -> %d\n", a);
}

```

```

        ret_val = _USUALLY(a >= 0) ? a : -1;
        return(ret_val);
    }

```

The `if (_RARELY(a < 0))` statement causes the compiler to lay out the code in such a manner that the `printf` is not the fall-through case. The code branches to the `printf` in the rare case that `a < 0`.

Similarly, the `_USUALLY` statement causes the code that sets `ret_val = a` to be the fall-through, usually-taken case. The code will branch and set `ret_val = -1` only when the expression `(a >= 0)` is not true.

The purpose of using these intrinsics is to eliminate branches. The elimination of branches is an important optimization, especially when running on a Pentium-Pro system.

Developers of performance-critical code are encouraged to make use of these functions. Attention should be paid to the most heavily executed conditional statements, as they will obtain the greatest performance benefit by using `_USUALLY/_RARELY`. Note that loop-control expressions (such as "`i<4`" in the loop within `main()` above) would not normally benefit from the use of a `_USUALLY` or `_RARELY` intrinsic, since both the compiler and the Pentium-Pro processor assume that loops normally branch back to the top instead of falling out.

Note that if code is meant to be portable so that it can be compiled with a compiler that does not support these calls, then the preprocessor defines from the above example must be included in the program. This causes the `_USUALLY()` and `_RARELY()` calls to be ignored by compilers which do not recognize them.

Packed and Unpacked Structures

Whether High C aligns structure members is implementation-dependent. Toggling *Align_members* aligns any structure definition occurring within any program region in which the toggle is on. Also see the pragma “Align_members” and “Pack” sections in Chapter 3, “Compiler Pragmas”.

You can also use the new keywords `_packed` and `_unpacked` to specify alignment on a per-structure basis. For example:

```
_packed struct {char c; int i;} x;
    /*Packed, as by default.      */
_unpacked struct {char c; int i;} y;
    /*Unpacked, not the default.  */
struct {char c; int i;} z;
    /*Packed, as by default.      */
#pragma Off(Align_members);
struct {char c; int i;} w;
    /*Unpacked, as by default.    */
```

Note: To comply with the i386 ABI, always use default packing.

__HighC__ Predefined Macro

This macro is predefined to be 1 for all implementations of High C unless compiling in ANSI mode. It can be used to exclude code that depends upon non-ANSI-supported features of High C.

Calling-Convention Attribute Specified with Function Declaration

In addition to the normal High C way of using the **Calling_convention** pragma, we have introduced a new calling-convention specification syntax that makes it easier to compile programs acceptable to Microsoft C; specifically, Windows applications, which contain lots of mixed-model programming.

The syntactic construct

```
_CC(E)
```

where E, an arbitrary expression, is permitted where `const`, `volatile`, `_Near`, and `_Far` are permitted. E must be a calling-convention expression, of the same sort permitted in the **Calling_convention** pragma.

For example:

```
_CC(_CALLEE_POPS_STACK | _SAVE_REGS)
```

specifies the Microsoft Pascal calling convention: callee pops the stack, non-volatile registers are saved if the routine uses them, and parameters are pushed from left to right (right to left requires the `_REVERSE_PARAMS` attribute).

The return type of a function, if it has `_CC` attributes, reflects those attributes to the function of being defined.

For example:

```
#define MSPASCAL _CC(_CALLEE_POPS_STACK | _SAVE_REGS)
int _Far MSPASCAL f() { . . . }
```

`f()` is a function returning a “far int with calling convention `_CALLEE_POPS_STACK | _SAVE_REGS`”. The attributes of the return type reflect onto the function being defined, so `f()` is really a “far function having calling convention `_CALLEE_POPS_STACK | _SAVE_REGS` and returning an int.”

Multiple `_CC( . . . )` specifications can appear in a single declaration; the effect of `_CC(e1) _CC(e2)` is the same as that of `_CC(e1 | e2)`.

The following restriction applies: The `_CC` syntax applies only in the presence of a function declarator. The following does not work:

```
typedef MSPASCAL x;
x f();
```

The MSPASCAL is lost.

The set of `_CC` attributes is cumulative in the specifiers. That is, if more than one `_CC` is specified, the final convention is the bit-wise OR of all the constituents. A calling convention specified by `_CC` completely overrides that given by a **Calling-convention** pragma.

Note that the calling convention present on a declaration of a function overrides the convention present on a subsequent definition, without any warning from the compiler. Thus in:

```
void MSPASCAL f();
void f() { . . . }
```

`f()` will have the MSPASCAL calling convention. In

```
void f() { . . . }
void MSPASCAL f();
```

`f()` will not have the MSPASCAL calling convention.

Note: To comply with the i386 ABI, use only the default calling convention. Libraries are built with the default, always use it when calling functions in them.

Shiftjis Kanji Support

The High C compiler supports double-byte Shiftjis Kanji characters in C strings. To engage the support, specify the **-Hkanji** on the driver command line or in the driver configuration file.

The Shiftjis Kanji support consists of a different algorithm to interpret a C string. In reading a C string from left to right, any occurrence of a character C1 in the range 0x80 . . 0x9f or 0xe0 . . 0xfc is considered to be the first character of a two-byte Shiftjis Kanji character. The character C2 that follows C1 is taken without interpretation. Thus C2 could be \, without confusing it with the standard C escape sequence.

This method of Shiftjis Kanji support is a Japanese standard. The ANSI Standard now dictates that character strings of the form L “XXXX” denote multi-byte character strings.

Compilation of programs using EUC Kanji do not require any special option on the driver command line or in the driver configuration file.

Direct I/O to Hardware Ports

Twelve intrinsic functions support direct generation of the 80x86 I/O port instructions. These functions are declared within the compiler as follows:

```
void _insb (int port, void * buf, int count);
void _insw (int port, void * buf, int count);
void _insd (int port, void * buf, int count);
void _outsb (int port, void * buf, int count);
void _outsw (int port, void * buf, int count);
void _outsd (int port, void * buf, int count);

unsigned char _inb (int port);
unsigned short _inw (int port);
unsigned long _ind (int port);
void _outb(int port, int data);
void _outw(int port, int data);
void _outd(int port, int data);
```

The first six functions perform stream I/O to the port. `_insb` translates to “load es:di with buf; mov dx, port; mov cx, count; rep outsb”. Similarly with the other stream instructions. These intrinsics are available only for 80286, 80386, and 80486 processors.

The last six functions perform single I/O operations to the port. `_inb` translates to “mov dx,port; in al, dx”, with the value in al the result of `_inb`. Similarly, `_inw` inputs a word into ax, and on the 80386 or 80486, `_ind` inputs a doubleword into eax. `_outb` translates to “mov dx, port; mov al, data; out dx,al”, and similarly for `_outw` and `_outd`.

In-Line Placement of Bytes of Code

`_inline(e1, e2, ... en)` places the least significant byte of constant expressions `e1, e2, ... en` directly in the instruction stream. This can be used, for example, to insert a CLI or STI instruction directly in an interrupt routine.

For example:

```
#define CLI _inline(0xfa)
#define STI _inline(0xfb)
CLI;      // Disable interrupts.
```

ASM and Direct Register Assignment for Variables

`_ASM` and direct register assignment provide a more powerful facility than `_inline`.

`_ASM` is a predefined function whose single string-constant argument is inserted in the assembly output of the compiler at the point of the `_ASM` call. Many such uses involve no registers and serve to change the status of the machine, such as clearing or setting interrupts:

```
_ASM("cli"); //Clear interrupts.
```

The optimizer normally assumes nothing about registers that can be used in `_ASM`. This means that the optimizer can use registers that you intended for your own purpose in `_ASM`, defeating the `_ASM`. Direct register assignment for variables is a method for communicating to the optimizer that you need registers for your own use.

In some implementations of High C, a register can be directly assigned to a variable by the following syntax in the variable's declaration:

```
== integer-constant-expression
```

For example:

```
int i == 5;
```

where 5 denotes `eax`. Furthermore, the size of the variable affects how many registers are used. Variables whose size does not exceed `4 * sizeof(int)` are supported; consecutive registers are allocated.

For the purposes of illustration, assume that for expression `E`, the register denoted is `E`. The following is a reasonable use of `_ASM`:

```
#define REG(i) (i) //For notational convenience.
int i -- REG(5), j == REG(7);
_ASM ("addl %eax, %ecx"); //j+=i;
```

The optimizer assumes that an `_ASM` statement reads and writes all registers directly assigned to variables whose scope contains the `_ASM` statement. In this example, the optimizer will be sure not to use registers 5 and 7 for other purposes.

Although the optimizer attempts to avoid using the registers assigned to variables, this may not be possible when registers are assigned dedicated purposes. If you assign a member of this subset to a variable, the contents of that register might be destroyed while building an argument list for a function call.

Direct register assignment can be used for communicating between functions when the register assigned is a volatile register (that is not saved and restored over function calls):

```
int i == REG(6);          void (f) {
i = 7          int i == REG(6);
    f();          if (i == 7) . . .
```

If nonvolatile registers were used instead, any assignment to the nonvolatile register in a called function would have no effect because the register would be saved and restored within that function. One-way communication, from caller to callee, is thus possible with nonvolatile registers.

High C supports `_ASM` and direct register assignment, with the register denoted by the expression explained in the following table.

Table F-2. Register Denoted by Expression

Processor	Register Denoted by Expression E	
Intel 80386/80486	ebx	for E = 1
	esi	for E = 2
	edi	for E = 3
	ebp	for E = 4
	eax	for E = 5
	edx	for E = 6
	ecx	for E = 7
	esp	for E = 8
	cs	for E = 10
	cs	for E = 11
	es	for E = 12

Processor	Register Denoted by Expression E	
	fs	for E = 13
	gs	for E = 14
	ss	for E = 15
	st(E-16)	for E in 16 .. 23

Restrictions

You can not reliably use a target machine's branch instructions within `_ASM`, because you might introduce in the program flow graph paths of which the compiler is unaware. Such paths can cause the compiler to generate incorrect code. Branch instructions should be contained within a single section of `_ASM` code, not intermixed with C. For example:

```
// Good example.
// ... Some C code here ...
// Here we make up a pretend assembly language:
_ASM(" cmp %eax,%ecx ");
_ASM(" je lab ");
_ASM(" movl 0, %ecx ");
_ASM("lab: ");
// ... More C code here ...
```

Using a branch and label across C code, as shown in the following example, is dangerous and should not be attempted:

```
// Bad example.
int i;
i = 1;
_ASM(" jmp lab ");
i = 3;
_ASM("lab: ");
int j = i; // Compiler assumes i = 3, not 1.
```

Furthermore, based on control flow analysis, the compiler may discard `_ASM` statements that appear dead:

```
_ASM(" jmp lab ");
return;
_ASM("lab: ");
_ASM(" movl %eax,%edx ");
```

The two `_ASMS` after the return are discarded because they are dead (code following a return; is never executed).

It is not reasonable to turn off dead code elimination, because the compiler uses it to eliminate extra compiler-inserted code that assists in other optimizations. Nor is it reasonable to assume that each `_ASM` breaks up a basic block, because the compiler would have to further assume that each `_ASM` can be branched to from every other `_ASM`, and the connectivity in the flow graph would result in terrible code for the C constructs near the `_ASMS`.

Use `_ASM` sparingly, and never to alter control flow across C constructs.

short/long enum

In a declaration of enum type, the adjectives short or long can be used. The former indicates that the enumeration type should use as little space as possible. The latter indicates that the enumeration type should have size equal to `sizeof(int)`. The former case is the default unless overridden by the toggle *Long_enums*.

For example:

```
short enum se (red,white,blue); //sizeof == 1
long enum le (alpha,beta,gamma); //sizeof == sizeof(int)
```

Note: To comply with the i386 ABI, enum data types must remain integers (as they are by default).

ANSI Preprocessor Numbers

ANSI preprocessor numbers have been implemented in High C. The lexical syntax for a number has been changed to the following:

```
'.'? Digit (Letter | Digit | '.' | ('e'|'E') ('+'|'-'))*
```

This means that 3 and 3.3f.e+e- are both valid preprocessor numbers.

Preprocessor numbers allow you to construct numbers out of their constituents with token pasting; for example, pasting 3e to 10 results in the floating point number 3e10 (see the “String Replacement (#) and Token-Pasting (##) in Macros” section earlier in this appendix). After all preprocessing is done, the compiler diagnoses any preprocessor numbers that are not valid numbers. So the 3.3.f.e+e- example given above will be diagnosed if it is not removed from the token stream during preprocessing, either by simple deletion or by string replacement.

Preprocessor numbers affect the High C extension .. that has been present in High C since its inception; namely, they invalidate it. For example, case 3..5: is now a single case with the value 3..5, which is a valid preprocessor number but an invalid number. Therefore, to protect the original .. extension, High C does not allow .. in a preprocessor number except when compiling in ANSI mode.

Switching to a New Base

Preprocessor numbers allow a new High C extension. High C recognizes a second or later x in a number beginning with 0x as a switch to a new base for interpreting the values after the x. The currently evaluated number becomes the value for the new base. In this way you can specify binary numbers or any other base up to 16 in High C:

```
0xfffe    // As usual.
0x2x1001  // Decimal 9 (binary number)
           // 0x2x1001x20    // Base 9 number with value 18
           // (20 base 9).
           0x10xfffe    // Same as 0xfffe since 0x10 is 16.
```

The extension is disabled if compilation is done in ANSI mode.

Iterators

This section describes a powerful language construct called an iterator. An iterator is a new kind of function that supplies values for the variables of an iterator-driven for loop. The for loop body is executed each time the iterator produces values for the for variables using a new predefined function `yield`, as discussed below.

The major benefit of iterators and iterator-driven for loops is that the algorithm to determine the values of the for variables for each loop iteration is defined separately in an iterator rather than being exposed in the Standard C for loop construct itself. The iterator may be invoked in many such for loops.

Iterators originated in CLU, a programming language from MIT that was designed to promote the use of abstractions in program construction. Iterators and iterator-driven for loops are not a part of Standard C. They have been made available in High C because they help programmers produce readable, easily modified code.

For example, in Standard C, you might write the following:

```
int i;
for (i = 1; i <= 10; i++)
    printf("%d squared is %d\n", i, i*i);
```

Here is how this loop could be written using an iterator:

```
void Upto (int o, int Hi) -> (int) {
    int i = Lo;
    while (i <= Hi) yield(i++);
}
for i <- Upto(1,10) do
    printf("%d squared is %d\n", i, i*i);
```

The sequencing from one number to another (1 through 10 in this example) is programmed once in the `Upto` iterator. The syntax for `i <- Upto(1, 10)` starts the iteration. Each time the iterator `Upto` calls the predefined function `yield` with a value, that value is substituted for `i` in the body of the for loop, and the body is executed. When execution of the body is finished, control is returned to a point immediately after the `yield` and `Upto` continues its while statement.

Each call to `yield` causes the for loop body to be executed once. When `Upto` is finished yielding, it returns just like a regular C function, and the invocation of `Upto` (and therefore the for loop itself) is complete. Control then passes to the statement after the for loop.

The syntax `-> (int)` in the header of the iterator definition is the only thing that distinguishes it from a normal function. After the `->` appears a prototype-form parameter list specifying the types of the results yielded by the iterator (more on multiple yields later). Here `Upto` yields an `int` (one for each execution of the for loop body). Within the definition of an iterator, the predefined function `yield` is defined. Outside an iterator, you can use the name `yield` for any other purpose.

Some Uses for Iterators

Iterators are quite general, but they are probably the most useful for iterating over each element in some set. The set is often the entire contents of some data structure, but can be restricted to those elements in a data structure that meet certain conditions. The set may not be stored in a data structure at all; for example, an iterator could be used to provide the prime numbers up to 100 by computing and yielding them.

The following sections present some programming situations where you might want to consider using iterators instead of regular for loops, and introduces a few of the programming problems that can be solved with iterators.

List Processing

Here is a loop that processes each element in a list:

```
for e <- each_element(list) do {
... // Process element e.
}
```

In the loop is the single for variable. Each time through the loop, `e` is given a value yielded by the iterator `each_element`. (In this case, you might expect `e` to have a different value each time throughout the loop, but this is not always so.) `e` is really a parameter to the loop body. It is not visible outside the loop body and is instantiated anew each time through the loop.

Assuming the following declaration of an element:

```
typedef struct element {
... // Some data.
    struct element *next;
} *element;
```

The iterator `each_element` might look like the following:

```
void each_element (element e) -> (element) {
while (e != 0) {
    yield(e)
    e = e->next;
}
}
```

Each time the predefined function `yield` is called, control passes to the `for` loop that invoked `each_element`. The `for` variable `element` takes on the current value of `e` for that pass through the loop body. At the bottom of the `for` loop, control returns to `each_element` at the statement following the call to `yield`.

Changing the Implementation of a Data Type

Suppose in some program you make heavy use of a sorted linked list. If you want to speed up your code, you might replace the list with a binary tree. That would mean not only replacing the code that implements the list (the `insert`, `delete`, and `sort` functions, for example), but also finding all the places in the program where the list gets traversed. You would need to replace all the `for` loops that are some variation on the following:

```
for (listptr = head; listptr != NULL;
listptr = listptr->next) {
    ... // Use list element.
}
```

You might traverse the list several places in your program, and you would have to change them all. On the other hand, if you use an iterator, you would still have to replace the code that implements the list, but you would not have to touch the code that traverses the list. Your loops would look something like the following (both before and after you changed over to a binary tree):

```
for listptr <- traverse_list(head) do {
    ... // Use list element.
}
```


Traversing a Tree

A for loop is a natural way to conceptualize getting each item in a data structure. You might write pseudo-code for some loop that processes every node in a tree, such as the following:

```
for each node in tree {
... // Process node.
}
```

However, when you actually write such a loop in C, probably you would not use a for loop. Trees are naturally traversed recursively, and C's for loop construct cannot naturally express a recursive computation. However, a High C iterator-driven for loop can express recursive computations as easily as any other kind; the resulting code looks almost identical to the pseudo-code.

The iterator `each_node` can be recursive. Thus, a recursive algorithm can easily be used to generate values for variables.

Recursion and Code Clarity

If it seems to you that straightforward loops have been made complicated, consider a different problem: Iterate through the nodes of a binary tree, where the tree is implemented using a data structure that, for each node N, has a pointer to the left and right subtrees of N.

The obvious way to obtain the nodes of such a tree is by a simple recursive tree walk. However, this is not possible using Standard C for loops. Imagine having to translate the following pseudo-code into C:

```
Process the tree:
do some stuff to the tree
  for each node in the tree
    print the node
  do some other stuff to the tree.
```

Because you cannot use a for loop to get the nodes of the tree, you would probably package the node-printing process into a routine and pass that routine to be executed by a tree-walking routine:

```
typedef struct node {
... // Some data.
```

```

    struct node *Left, *Right;
    // Left and right subtrees.
    } *Node;
    void Print_node (Node N) {
        printf("Node is ");
        ... // Code to print a Node.
    }
    void Each_node(Node N, void Doit_toit(Node N) {
    if (N == 0) return;
        Each_node(N->Left,Doit_toit);
        Doit_toit(N);
        Each_node(N->Right,Doit_toit);
    }
    Process_the_tree(Node Root) {
        ... // Do some stuff to the tree.
        Each_node(Root,Print_node);
        ... // do some other stuff to the tree.
    }

```

Rearranging the computation this way allows the recursive tree walk to occur, but there is a cost. The simple for loop in the pseudo-code clearly expresses that a computation is being done on each element of the tree data structure; this is no longer as apparent in the body of `Process_the_tree`. To find out what is being done, you now have to look outside `Process_the_tree`, in `Print_node`. In this simple example, the choice of good names assists a great deal in promoting understanding of the code, but when you are dealing with real problems, the code is more complex and the names are usually not perfectly explanatory.

Here is a solution to the same problem, using iterators:

```

void Each_node(Node N) -> (Node) {
    if (N == 0) return;
    //Walk the subtrees.
    for L <- Each_node (N->Left) do yield(L);
    yield(N);
    for R <- Each_node(N->Right) do yield(R);
    }
    Process_the_tree(Node Root) {
        ... // Do some stuff to the tree.
        for Node <- Each_node (Root) do {
            printf(Node is ");
            ... // Code to print a Node.
        }
        ... // Do some other stuff to the tree
    }

```

Notice that the code for `Process_the_tree` mirrors the pseudo-code extremely closely; yet, unlike the Standard C version, the algorithm that determines the successive values of `Node` in the for loop body is recursive. This is a major increase in expressive power. Algorithms can be expressed in a more natural fashion, making the code easier to write, understand, and modify.

Note that there is no claim that computational power has increased. C is already Turing-machine equivalent.

The body of `Each_node` is not very elegant (or efficient) as written. The recursive calls do nothing but re-yield a result already yielded at a deeper level of recursion. Using nested functions (another High C extension) and the fact that calls to yield can occur within functions nested within iterators, we can produce the following elegant version:

```
void Each_node(Node N) -> (Node) {
void P(Node N) {
    if (N == 0) return;
    // Walk the subtrees.
    P(N->Left);
    yield(N);
    P(N->Right);
}
p(n);
}
```

Work done by the programmer in the Standard C version (packaging the body of the pseudo-code for loop as a function) is done instead by the High C compiler in the iterator version. This is appropriate; after all, compilers were invented to promote more easily understandable and modifiable languages. Why make a programmer do what the compiler can do?

Replacing Macros with Iterators

Another place an iterator can be gainfully used is when the iteration algorithm, although not recursive, is complex. Often, in this circumstance, a macro is designed to make up for the shortcomings of the C for loop. Consider this example: The problem is to sequence through the objects that overlap a given object `obj` in memory. Prior to the addition of iterators, a macro was required to simulate the following iteration:

```
#define for_each_overlapping_object o,obj,p) {\
```

```

struct obj_entry *_op = &objtab(obj);\
obj_class_type_class = _op->class;\
long _len = op->len;\
long _disp = _op->disp;\
ushort _word = _op->un.word;\
bool is_deref = (ea_DEREF &_op->flags != 0);\
bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags) != 0;\
register struct obj_entry *p; int o;\
for (o=1, p=objtab+1;o<=last_object;o++,p++) {\
if (is_deref && \
    ((p->flags&ea_DEREF) || \
     (p->flags& (ea_ADR|ea_GLOBAL))) !=0 && \
    p->xlen >= _op->xlen && \
    _op->disp < p->xlen && \
    is_adr && (p->flags&ea_DEREF) !=0 && \
    p->xlen <= _op->xlen || \
    p->class == _class && \
    (p->un.word == word || \
     (p->flags*_op->flags&ea_TEMP)) && \
    (p->disp <= _disp && p->disp+p->len > _disp || \
     _disp+_len > p->disp && _disp < p->disp)) {

```

It was invoked as follows:

```

object_index obj;
. . .
for_each_overlapping_object(o,obj,p)
. . . do things with o and p . . .
}}} // Required to match {s in macro.

```

The #define defined the variables o and p, whose names were arguments to the macro.

With this approach, an enormous amount of code is reproduced each time the macro is invoked; and every time a change is made to the macro, every module where it appears must be recompiled. An iterator requires much less maintenance; when a change is made, only the iterator itself and not its client for loops need be recompiled. Another advantage is that the iterator can declare the types of its parameter and yielded results, providing improved type checking at the iterator invocation.

```

void Each_overlapping_object(object_index obj)
-> (int o, struct obj_entry *p) {
    struct obj_entry *_op = &objtab[obj];
    obj_class_type_class = _op->class;
    long _len = _op->len;
    long _disp = _op->disp;
    ushort _word = _op->un.word;
    bool is_deref = (ea_DEREF) & _op->flags != 0;
    bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags != 0;

```

```

register struct obj_entry *p; int o;
for(o=1,p=objtab+1;o<=last_object;o++,p++) {
if (is_deref &&
((p->flags&Iea_DEREF) ||
(p->flags&(ea_ADR|ea_GLOBAL)) !=0 &&
p->xlen >= _op->xlen &&
_op->disp < p->xlen) ||
is_adr && (p->flags&ea_DEREF) != 0 &&
p->xlen <= _op->xlen ||
p->class == class &&
(p->un.word == word ||
(p->flags&_op->flags&ea_TEMP)) &&
(p->disp <= _disp && p->disp+p->len > _disp ||
_disp+_len > p->disp && _disp < p->disp)) {
yield(o,p);
}
}
}
}

```

It is now invoked as follows:

```

object_index obj;
. . .
for o,p <- Each_overlapping_object(obj) do
. . . do things with o and p . . .

```

Note that the iterator yields two results for each execution of the for loop body: the int o and the struct obj_entry *p.

The following section contains a more precise specification of iterators and iterator-driven for loops.

Syntax and Constraints

To invoke an iterator **I** for n variables, write the following:

```

for Name1, Name2, ... Namen <- I(E1,E2, ... En) do
. . . // Body of for loop, using the namei.

```

Iterators yield one or more values. The declarative syntax for both the iterator input arguments and yield types follows that of the input arguments to normal functions. An iterator **I** is declared as follows:

```

void I (Formal_parameter_list1) ->
(Formal_parameter_list2) {
. . . // Body of iterator, with calls to yield.
}

```

Arguments

The iterator's `Formal_parameter_list1` has exactly the same constraints and semantics as the formal parameter list of other functions, except that we additionally require that this list use the ANSI prototype syntax.

Old-style C function parameter declarations are not permitted. When the iterator is invoked in a for loop, the expressions passed to it must satisfy the same constraints as those of expressions passed to a function, given the same parameter list, and the parameters are passed in the same ways.

The `Formal_parameter_list2` must also use the ANSI prototype syntax. Here, however, the names of the parameters may be omitted, just as they may be omitted when a function is declared and not defined.

The `Formal_parameter_list2` is called the yield list. The for variables of a for loop that invokes the iterator take on values of the types specified in the yield list, respectively. The iterator supplies the values through a call to the function `yield`, which is defined in the body of each iterator:

```
yield(E1,E2, . . . En);
```

where the following constraint must be satisfied:

If `void yield(Formal_parameter_list2);` is a valid function declaration, then `yield(E1,E2, . . .En):` must be a valid call to that function.

Alternatively, the syntax of High C's named parameter association can be employed to yield the values, just as in a function call. The `Formal_parameter_list2` must contain the names of the parameters (see the section entitled "Named Parameter Association" earlier in this appendix). However, even if that notation is not used, it is useful to name the "yield parameters" for the sake of the documentation.

Like other functions, iterators can use the `...` notation to specify that they can take a variable number of arguments. The yield formal parameter list can also use `...`, requiring the use of the `va_arg` macros within the for loop body to access the remainder of the expressions yielded.

Semantics

Each time an iterator executes a yield statement, the body of the for loop is executed with its *i*th for variable, assuming the value of the *i*th expression yielded, for all $1 \leq i \leq m$. When the iterator returns (just as a function may return), the for loop is terminated. A break or goto from the for loop body also terminates the iteration.

The semantics of iterators and for loops that invoke them can be specified precisely in terms of an implementation using nested functions, as follows.

The compiler bundles up the body of each iterator-driven for loop and turns it into a function. Each so-bundled function is passed as an extra parameter to the iterator; the iterator is called once per for loop and receives as a parameter the function that used to be the body of that for loop. Each call to yield within the iterator is translated to a call to the function that is its extra parameter.

More formally, the meaning of the following:

```
void I (Formal_parameter_list1) ->
(Formal_parameter_list2) {
    . . . // Calls to yield in here.
}
```

and

```
for Name1, Name2, ... Namen <- I(E1, E2, ... En) do
    . . . // For loop body here.
```

is precisely the same as the meaning of the following:

```
void I (void yield (Formal_parameter_list2) !,
(Formal_parameter_list1) {
    . . . // Calls to yield in here.
```

and

```
{//Would-be for loop:
void For_loop_body (Formal_parameter_list3) {
    . . . // For loop body here.
}
I (For_loop_body, E1, E2, ... En);
}
```

where Formal_parameter_list3 is Formal_parameter_list2 but with the names Name1, Name2, ... Namen replacing the parameter names (if given) in Formal_parameter_list2.

Note that the calls to yield in the iterator I become calls to function I's function parameter name yield - the syntax is the same in both cases. Also, the for loop body is made into a function and passed as an extra first parameter to the function I. Calling function I is what start the loop; I's return terminates the loop, barring a goto out of the loop body.

It is the function I that does the iterating. That can occur by way of a contained loop, or by I calling itself recursively as it traverses some data structure, for example. Each time it wants to yield something to the for loop body, it does so by calling its yield parameter and passing the values of the for variables to the next iteration of the loop.

The ! declaration guarantees we can pass a nested (non-global) function to the iterator. Because the for loop is guaranteed to lie within a function, the for loop body, when transformed into a function, is guaranteed to be a nested function. For example:

```
void Primes(int LO,int HI) -> (int ThePrime) {
// We name the yield result.
// Yield the primes in the interval Lo..Hi.
int I,J;
extern double sqrt(double);
for (I=Lo; I<=Hi; I++) {
// Ask if we can divide I evenly:
for (J=2; J<=sqrt(I); J++)
if ((I/J)*J == I) goto Composite;
yield(I); // I is prime.
Composite: ;
}
}

for I <- Primes(1,100) do printf("%d is prime.\n",I);
```

This example has the same semantics as the following example.

```
void Primes (void yield(int ThePrime) !,int LO,int HI) {
// We name the yield result.
// Yield the primes in the interval Lo..Hi.
int I,J;
extern double sqrt(double);
for (I=Lo; I<=Hi; I++) {
// Ask if we can divide I evenly:
for (J=2; J<=sqrt(I); J++)
```



```

if ((I/J)*J == I) goto Composite;
yield(I); // I is prime.
Composite: ;
}
}

void For_loop_body(int I) {
printf("%d is prime.\n", I);
}
Primes(For_llap_body, 1, 100);
}

```

Predefined Function Yield

Because the types yielded by an iterator are described by a `Formal_parameter_list`, there is no restriction on the types except that they must be types that can be passed to a normal function. Thus, an iterator can yield integers, structures, functions, and even iterators. An iterator that computes the strongly connected components of a graph and an example of its use are provided in High C distributions in the file *analyze.c*; this iterator's yield is itself an iterator that yields one set of strongly connected components. No one has yet accepted our challenge to rewrite this sample in Standard C.

Advantages and Disadvantages

Advantages and disadvantages of iterators are discussed in this section.

Reusable Code

The major benefit of iterators is that the algorithm to determine the values of the for variables for each loop iteration is defined separately in the iterator. Unlike the Standard C for loop, the iterator can do an arbitrary amount of computation and can automatically maintain its environment across passes through the body of the loop, because it suspends, rather than returns, when it provides the loop body with values. Once the iterator is written, it can be invoked by as many loops as desired without repeating the code that provides the values used within the loop (as compared with using many copies of standard for loops, which involve repeating the code for each loop).

Information Hiding

The term information hiding comes from the literature in structured programming and refers to the placement of information only where it is needed and keeping the information from where it is not needed. For example, a module implementing a tree data type may wish to hide all the details of traversing trees within functions defined in that module. In Standard C, however, traversing the tree generally means exposing the tree data structure in any for loop sequencing through nodes of the tree. Instead, iterators can be used to localize the sequencing techniques and their required data structure access to within the tree module itself. The for loop need only invoke an appropriate iterator and need not be concerned with the representation of trees.

Information hiding promotes better and more easily maintained programs, and reduces recompilation of modified programs. Iterators promote information hiding.

Execution Speed

The major drawback of iterators is that they are slow, compared to standard for loops, because they involve the overhead of function calls. Speed is lost by placing the computation of the iteration in a separate function. A loop that executes n times involves $n+1$ function calls related to loop overhead. The speed loss occurs not only due to the function linkage, but also because the body of the for loop becomes a nested function, and any variables in the body's parent accessed from the body can not reside in high-speed machine registers.

For example:

```
int Sum = 0;
for I <- Upto(1,10) do
    Sum += I;
    printf("Sum of 1 to 10 is %d\n",Sum);
```

This is implemented as:

```
int sum = 0;
void Body(int i) {Sum += I;}
    Upto(Body,1,10);
    printf("Sum of 1 to 10 is %d\n",Sum);
```

Here, Sum cannot reside in a register because it is accessed from nested function **Body**. Thus all accesses to Sum will be slowed. Therefore, if extreme speed is required, do not use iterators, especially for simple for loops where the iteration algorithm is simple and there is no need for hiding it away in one place in an iterator.

Conclusion

Iterators allow you to express some computations more naturally by virtue of recursion, and they can make your code easier to maintain because there are fewer places where code must be modified if there is a design change.

Recursion is possible with iterators because the function that drives the for loop does not have to return.

If you use for loops to access the elements of a data structure, iterators allow you to change the implementation of the data type without changing the implementation of the for loops. With an iterator, each for loop is driven by an independently specified computation that can maintain its environment across invocations of the loop. Without an iterator, the syntax accessing the structure is inside each for loop; therefore, each for loop must be modified, since the loops contain code specific to the data structure.

Prototypes

X3J11 provides an alternate form for specifying the parameter types in function declarations. When functions declared in this form are called, the types of the arguments must be assignment compatible with the types of the formal parameters, and as in assignment, any necessary conversion to the formal parameter type is applied.

In contrast, KR specifies that the types of the declared parameters (if available) are irrelevant at the function call, and instead that default conversions are required: chars are widened to ints and floats to doubles. But this can lead to incorrect and unpredictable results when the type of the value passed is not compatible with the declared type; for example:

```
float Min(f1,f2) float f1,f2; {
return f1 < f2 ? f1 : f2;
}
int i; float f,fmin;
fmin = Min(i,f);
```

Here, an int i is passed to Min instead of a double. In addition, if sizeof(int) != sizeof(double), the f2 parameter to Min is not passed in the correct location on the stack. (Recall that doubles are always passed as parameters in KR, never floats.)

In “prototype” syntax, the specification of a parameter name and its type are not separated and appear more like an “old C” declaration.

The parameter declarations are separated by commas, just as in old C parameter lists:

```
float Min(float f1, float f2) {
return f1 < f2 ? f1 : f2;
}
int i; float f,fmin;
fmin = Min(i,f);
```

With the prototype syntax, two things happen. First, float values are passed, not doubles. This avoids the expensive widening-at-call and shortening-at-function-entry cost of KR. Second, the int i is converted to float before it is passed, just as if i were being assigned to a float variable.

Notice the repetition of float in the parameter: no factoring of declarations is permitted, as in float f1,f2 due to syntactic constraints.

A prototype declaration can be used anywhere an old C declaration is allowed. In addition, the parameter names can be omitted in declarations that are not definitions. For example:

```
float Min(float, float);
```

Prototype functionality is new, designed to allow some type checking across separate compilation units.

Loop Unrolling

The following topics are covered in this appendix:

- Introduction to Loop Unrolling
- Selecting Loops for Unrolling
- How Loops are Unrolled

Introduction to Loop Unrolling

This appendix describes an optimization called loop unrolling (or loop unwinding). Loop unrolling is most effective on machines with a pipelined architecture.

If the next iteration of a loop does not rely on results of the prior iteration, two or more iterations can execute at the same time. This works only for loops with constant stride: those whose loop counter is incremented by a compile-time constant for every iteration. Only a small saving in execution time may be realized on machine architectures that do not have pipelined instructions. On machines with an instruction cache, loop unrolling can actually degrade performance if the unrolled loop exceeds the size of the cache.

Selecting Loops for Unrolling

Use pragma **Loop_factor** to specify criteria for selecting loops to be potentially unrolled. The syntax is:

```
pragma Loop_factor(loop_count, size);
```

where *loop_count* is the maximum number of times a loop will be unrolled, and *size* is the upper limit on the size of loops to be considered for unrolling. *size* refers to the number of intermediate-language instructions generated by the compiler; this number corresponds loosely to the number of individual machine instructions.

You can specify pragma **Loop_factor** any number of times, with different values for *size*. You can have any number of loop factors in effect at the same time, each with a different *size*; however, a pragma with the same *size* as a previous pragma replaces that previous pragma.

To disable a previously specified program, use pragma **Loop_factor(0,size)**, where *size* is the same as specified in the pragma you want to disable.

If you specify pragma **Loop_factor(n,0)**, *n* becomes the default number of times a loop is unrolled, regardless of *size*. If you also specify (for instance) pragma **Loop_factor(x,100)**, loops with less than 100 intermediate instructions will be unrolled *x* number of times, and loops with more than 100 intermediate instructions will be unrolled *n* number of times.

You can also specify loop unrolling on the compiler command line, with option **-Hunroll=loop_count,size** (see “Invoking the Compiler”). For example:

```
hc cwhetd.c -Hunroll=3,20 -Hunroll=4,10
```

is equivalent to:

```
pragma Loop_factor(3,20);
pragma Loop_factor(4,10);
```

How Loops are Unrolled

The loops to be unrolled must have certain characteristics, as well as meeting the selection criteria you specify with pragma **Loop_factor**. To be a candidate for unrolling:

- The loop control variable must not be modified in the body of the loop.
- A loop must have constant stride. That is, the controlling loop variable must increase or decrease by a constant amount each time around the loop.
- The resulting *loop_count* must be greater than 1. Otherwise, the loop is not unrolled.

The *loop_count* determines how many times a loop is unrolled. A general loop, that is, a loop for which the number of iterations is not known, is unrolled *loop_count* times, and then a remainder of the iterations (up to *loop_count* - 1) are executed by a following loop that is not unrolled.

A constant loop (one where the number of iterations *n* is known) that has less than *loop_count* iterations is unrolled *n* times, and no loop-test conditional jumps are performed. The same happens for constant loops with more than *loop_count* iterations but less than *loop_count* * 2 iterations. The following example illustrates what can happen for a constant loop with more than *loop_count* * 2 iterations.

Suppose you specify:

```
pragma Loop_factor(3,100);
```

and the following for loop occurs somewhere in your program.

Example for Loop

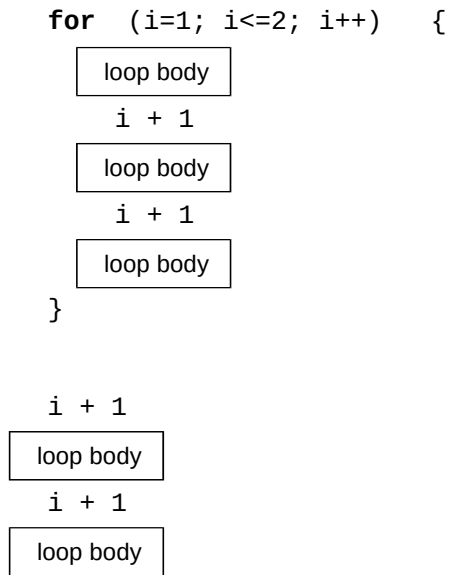
```
for (i=1; i<=8; i++) {  


loop body

  
}
```

If code generated for the body of the loop amounts to less than 100 intermediate instructions, the loop may be unrolled as shown in the following figure.

Example for Loop, Unrolled



Structure Report and Call Graph Facility

The following topics are covered in this appendix:

- Getting and Using a Call Graph
- Structure Usage Reporting

These topics describe:

- how to measure the data accessing and function calling behavior of a program
- how to make the most of what your measurements tell you

The new compiler command line option **-qs** adds two new tools to those provided by the **-qb** option. The **-qb** option specifies that you want the program compiled with basic block profiling probes inserted. The **-qs** option adds these probes just as **-qb** does. It also adds a list to every basic block in modules compiled with **-qs** that notes what functions are called from that block and what structure accessing activity that block performs. It only lists calls to functions whose names are known (that is, calls through a pointer are not listed). Post processing with a file that contains basic block execution count information (a *bbarc.out* file) that was produced by a program with modules compiled with the **-qs** option yields call graph and structure accessing information about the execution that produced the *bbarc.out* file.

If the **-qb** and **-qs** options are both used on a compiler command line, the last (rightmost) one overrides the first (leftmost) one.

Getting and Using a Call Graph

To get a call graph of the execution of a program compiled with **-qs**, run the **profcnt** post processor with the new **-c** command line option:

```
profcnt -c -o executable bbarcFile
```

The output goes to standard out with one line per caller/callee pair. The form of each line is

```
count caller callee
```

The unprocessed profcnt output may not always have all like-function caller/callee pair data on one output line. It is sometimes useful to pipe the profcnt output to the **cgsort(1)** utility.

Calls from functions in source files not compiled with **-qs** are not recorded, but calls from functions in source files compiled with **-qs** to functions in source files not compiled with **-qs** are recorded. For example, a call to the malloc utility might be shown as

```
38243 makeSymbol malloc
```

The lines are sorted by the count, so that the most frequently occurring control transfers appear at the beginning of the file.

Site Specific Inlining Using a Call Graph

Another new compiler command line options supports site specific inline substitution of the body of a function in place of the call. This facility is best used in conjunction with the intermediate compilation facility. (See Appendix I, “Intermediate Compilation”). Use this new option as follows:

```
cc ... -F=<callGraphFile> ...
```

The compiler's inline substitution module is normally controlled by three parameters:

- **-Hic=n**—This parameter says that it may inline only those functions which are called less than n times.
- **-His=n**—This parameter says the it may inline only those functions whose local stack needs are less than n bytes.

- **-Hit=n**—This parameter says that it may inline only those functions that have less than n intermediate language nodes.

The occurrence of a caller/callee pair in the callGraphFile tells the inliner to ignore these limits for any call from function caller to function callee. The normal inliner limitations other than the three listed above still apply. For example, the inliner can not put any function inline that uses a variable number of arguments, that uses built in function `_alloca`, or that calls an external function named `alloca`. (See Appendix E of the *MetaWare High C/C++ Programmer's Guide for Unix on 386-486-Pentium* for more information on inline substitution.)

Typically, you would not choose to force inline substitution at all of the caller/callee sites listed in a call graph file. You should edit the file judiciously, choosing only the hottest caller/callee pairs for site specific inline substitution.

You can construct a site specific inline substitution file manually if you like. The inliner ignores the count part of a line, so you need not write one. It also accepts lines that name only one function. A line like

```
hashFunc
```

specifies that the inliner is to replace every call to `hashFunc` with its body.

Function Reordering at Link Edit Time

The new **ld(1)** command line option **-zcallgraph=<file>** supports using a call graph file to tell **ld** how to place the functions of the program in the virtual address space. For this use, the counts are required. **ld** reads the file and places functions that call each other into clusters. It starts with the two functions named in the first call graph file line (normally the hottest). It puts them into the first cluster. It continues with successive call graph lines. If one of the functions on the line is in an existing cluster and the other is in no cluster, it adds it to that cluster. Callee functions precede caller functions in a cluster so that the P6 default branch prediction properly predicts that the call will occur (the P6 default branch prediction is that backwards branches transfer control and forward branches do not). Similarly, when the caller is found to be already in one cluster and the callee in another, **ld** places the callee's cluster before that of the caller.

If neither the caller nor the callee of a new line is in an existing cluster, **ld** creates a new cluster for them. If both the caller and the callee function are already in a cluster, **ld** orders the clusters so that the one in which the callee is placed precedes that of the caller in the virtual address space.

ld takes care not to split a function that is in a cluster across a page boundary. It aligns each cluster on a page boundary. If a cluster exceeds one page in size, **ld** divides it into multiple pages, taking care to put functions with hot calling arcs into the same page. When **ld** splits a cluster into multiple pages, it prints a warning message. It is prudent to run the **CacheOpt** utility on any executable for which this message appears to guarantee that the order of the pages in the virtual address space corresponds to that in the physical address space. (The L1 instruction cache is managed according to physical addresses. Unless **CacheOpt** is used, the multiple pages that result from one cluster may overlay each other in the instruction cache, defeating the very purpose of function reordering.)

When reordering the functions of a program that has been compiled with the **-qs** option, one should probably use the **-Hon=ncr_func_split** option when recompiling with the **bbarc.out** file to reorder the basic blocks of the program. The effect of the **-Hon=ncr_func_split** option is to divide every function into a hot and a cold part. The hot part contains all of the basic blocks that were executed during the profiling run(s). The cold part contains the blocks which were not executed during the profiling run(s). Only the hot parts are rearranged during function reordering.

When the **-Hon=ncr_func_split** option is used, the code is compiled using jumps from hot to cold blocks and back so that the cold code can be executed as needed in the executable that is produced. The beneficial effect from using this option is that the cache space occupied by each function is reduced to that needed only by the part that was executed in the profiling run(s).

Merging Call Graph Files

You may run a program that was compiled with **-qs** several times with different data sets. To merge the call graph files from several different executions, run the new **cgsort** utility: The output of

```
cat cg1 ... cgN | cgsort > mergedCG
```


in file *mergedCG* has one line for each caller/callee pair in the set of call graph files. The call count on each line is the sum of the counts for all lines for that caller/callee pair in the input file(s). The *mergedCG* file is sorted by the accumulated call counts.

Structure Usage Reporting

The **-qs** option also specifies that information about structure usage be noted for each basic block. Both static and dynamic (through a pointer) usage is noted.

Use the new **profcnt -s** option with the *bbarc.out* file that a program compiled with the **-qs** option writes when it completes execution to display the structure usage activity of the program.

```
profcnt -s -t N -o executable bbarcFile
```

The **-s** option adds structure report information to the execution profile information that profcnt writes to standard out.

Format of a Structure Usage Line

The form of a structure usage line is

```
structure name { member name[displacement] ... }
```

Each member name is preceded by a symbol that indicates whether its value was read, written, or both read and written by the basic block:

```
<   The member value was read.
>   The member value was written.
<>  The member value was read and written.
```

Each member name is suffixed by its displacement in the structure enclosed in square brackets. These displacements are the key to making the most of the structure usage information. When a hot basic block accesses structure members that are so far apart that they must be in different data cache lines, the structure becomes a prime candidate for reorganization. Moving the members close together in the structure will reduce the number of data cache lines required to support execution of the basic block. Often, the number of cache lines that are needed for all of the data that the block references can be reduced to one.

A typical structure usage report line is:

```
status { <count[0] >touched[4] }
```

Six profcnt Report Types

The value of **N** in a **profcnt** command line determines how the execution results are organized in the report. The structure report information is organized in the same way, interspersed with the execution profile lines.

- **N=0**—The report organizes the execution count information by basic block. Each line is preceded by the number of times the basic block was executed. The lines are sorted based on these counts. Structure usage and function calling information for each block follows the execution count line.
- **N=1**—The report organizes the execution count information by function name. Each function line is preceded by the number of times the hottest block in the function was executed. Structure usage and function call lines are preceded by the number of times the basic block in which they occurred was executed.
- **N=2**—This report type adds a line to the type 1 report for each basic block in the function. The execution count for the basic block is the first time on the line. Structure usage and function call information follows that for the block in which it occurred.
- **N=3**—The report organizes the execution count information by source module name. The execution count for the hottest basic block in the module precedes the other information on the module identification line. Structure usage and function call lines follow this line. Each begins with the execution count of the basic block in which the activity occurred. Lines are sorted by count within each module.
- **N=4**—This report is like a type 3 report with lines added for each function. Structure usage and function calling lines follow the line for the function in which the activity occurred.
- **N=5**—This report is like a type 4 report with lines added for each basic block. Structure usage and function calling lines follow that of the basic block in which the activity occurred.

getFieldUse Warning Message

Some programs define structures that end with arrays whose size is determined at run time. The structure declaration in these cases contains an array of size zero or one, but the runtime size is often much bigger. When a basic block refers to an array element that is beyond the extent of the declared array, the displacement into the array appears to lie outside of any declared member. In cases like this, the compiler issues a warning message like this:

```
getFieldUse: no entry for displacement 12(1037)
```

The 12 in the example is the displacement, and the 1037 is the compiler's internal type number for the structure. The compiler writes its internal type numbers and their meaning to standard out if you add option -

Hon=opt_input to the command line that includes the **-qs** option.

Intermediate Compilation

The following topics are included in this chapter:

- Building Intermediate Object Code Files
- Controlling Inline Substitution in Intermediate Object Code Files
- Site Specific Inline Substitution
- Building Libraries From Intermediate Object Files
- Building Normal and Intermediate Object Files in the Same Directory
- Environment Variables With the Intermediate Compilation Facility
- Output From a Command Line Using **-Hih** and **-Hit=64**

Intermediate Compilation, which is an easier way to accomplish the effects of wide compilation, has the following benefits:

- It can be used across multiple directories.
- It permits differing command line options for the modules that are compiled "together."
- It supports using library archives.
- It supports site specific inlining.

A wide compilation presents all of the source files that are named on the same compiler command line to the inline substitution module of the compiler at the same time. This permits cross-module passing of arguments in registers and inline substitution for all of the functions in the set of source files that are compiled together. After the inliner has finished, the compiler control program **cc** sends the resulting intermediate code to the optimizer one module at a time to produce the final set of object code files.

This is straightforward when all of the source files are in the same directory and use the same command line options. Intermediate compilation is a compiler feature that simplifies achieving the effects of a wide compilation when

- the source files are in multiple directories
- some of the desired modules come from library archive files (including standard libraries for which intermediate versions are available)
- different command line options are required for modules that need to be compiled together
- you need site specific control of some or all of the inline substitutions that occur

Building Intermediate Object Code Files

Intermediate compilation is specified by the **-ilobj** command line option. The effect of the option is to produce object files (**.o** files) that contain the compiler's intermediate code representation for the module rather than its normal final form. For example, the command lines

```
cc -c -ilobj ... dirx/x.c
cc -c -ilobj ... diry/y.c
cc -c -ilobj ... dirz/z.c
```

build intermediate files *dirx/x.o*, *diry/y.o*, and *dirz/z.o*. Each of the three command lines above can have different options. The part of those options that apply to the compiler's optimizer are stored in the intermediate object file and applied when the final the intermediate code is combined into an executable. The subsequent command line

```
cc -o xyz -ilobj dirx/x.o diry/y.o dirz/z.o
```

results in the intermediate code for *x.c*, *y.c* and *z.c* being presented to the compiler's inline substitution module at the same time. The temporary files that are the result of the inline substitutions and argument passing in registers that are produced by the inliner are each compiled separately with the correct optimizer options and are used to build the executable *xyz*.

If file *diry/y.o* in this example is a normal object code file, then only the intermediate code for *dirx/x.o* and *dirz/z.o* is presented to the inliner, and the normal code for *y.o* is linked with the final object code for *x.c* and *z.c* in *xyz*.

An intermediate object file has the same symbol table that the corresponding normal object file would have. It can be processed by all of the tools that operate on object files. It can be put into an archive by **ar(1)**. **dump(1)** and **nm** display its definitions and undefined symbols. When intermediate object files are given to **ld(1)** with the **-ilobj** option, **ld** resolves interfile dependencies just as it would with the corresponding normal object files. Intermediate object files are not small. Typically, an intermediate object file is up to six times the size of the corresponding normal object file.

Controlling Inline Substitution in Intermediate Object Code Files

Control of inline substitution is normally limited to specifying values for the three controlling options:

- **-Hic=*n*** The value of *n* tells the inliner that it may put inline only those functions that are called less than <number> times.
- **-His=<size>** The value of <size> tells the inliner that it may put inline only those function whose stack usage requirements are less than <size> bytes.
- **-Hit=*n*** The value of *n* tells the inliner that it may put inline only functions with less than *n* intermediate language nodes in their representation.

Inline substitution is inhibited by certain conditions under all circumstances. For example, a function that has a variable number of arguments or that makes a call to the built in function `_Alloca` (or to an external function named `alloca`) can not be put inline. (See Appendix E of the *MetaWare High C/C++ Programmer's Guide for Unix on 386-486-Pentium* for more information on inline substitution.)

A command line that builds an executable from intermediate object files can specify values for the three numerical inline substitution control limits. For example, the previous command line could have been

```
cc -o xyz -ilobj -Hit=128 dirx/x.o diry/y.o dirz/z.o
```

to specify that functions with up to 128 intermediate language nodes should be put inline in xyz. To see what inline substitutions are performed, add command line option **-Hih** to the command line. It is also generally useful to add the **-HIZ** compiler option to request that arguments functions that are not put inline be passed in registers.

Site Specific Inline Substitution

When you know that some of the program's function calls occur very often, you can override the normal inline substitution limits for those calls with the new **-F=<file>** command line option. You can get function calling information about the program by compiling it with the **-p** command line option that supports the **prof(1)** and **gprof(1)** profiling post processors. (See the man pages for **prof** and **gprof** for more details.) You can also get function calling information about a program by using the **-qs** option. See Appendix H, “Structure Report and Call Graph Facility”, for more information.

Each line in <file> has one of these two forms:

```
caller callee
callee
```

A caller callee line tells the inline substitution process to put function callee inline in function caller no matter whether its **-Hic**, **-His**, and **-Hit** parameters meet the limits that are in effect. A callee line tells the inline substitution process to put function callee inline at every call site. The occurrence of a callee in the file will not override the inherent restrictions on inline substitution like having a variable number of arguments. It only effects the call count, stack size, and node count limits.

To limit inline substitution to only those sites named in the **-F=<file>**, set one of the size limits to zero. For example, this command line will permit only the inline substitution specified in <file>:

```
cc -ilobj -F=<file> -Hit=0 dirx/x.o diry/y.o dirz/z.o
```

To see what inline substitutions are performed, add command line option **-Hih** to the command line. The output in Appendix A, “Diagnostic Messages” is from a compile using **-Hih** and **-Hit=64**. Another way to determine which function calls occur frequently in a program is to get a call graph file of that program's execution. (See “Structure Report and Call Graph Facility” for more information on how to get a call graph file.) The lines in a call graph file have the form:

```
count caller callee
```

When a call graph file is used as the file of the **-F=<file>** command line option, the inliner ignores the count field when processing the lines in the file. Typically, one would not use the entire call graph file, but would choose only the first few lines that indicate hot function caller/callee pairs.

Building Libraries From Intermediate Object Files

Many applications use archive files (libraries) to store some of the object code that constitutes the application. The files in these libraries can be intermediate object files. When the final step that builds the executable takes place, the intermediate files from the libraries are used. The intermediate code from those files is presented to the inliner just as though the library source files had been included in a wide compilation with the source files that do not correspond to library members. The optimizer options that were used when the intermediate files were produced are applied to the optimization phase of their compilation after the results of the inline substitution process is complete.

The above discussion applies to static libraries, either those that are specific to the application or standard libraries such as *libc.a* for which an intermediate form is available.

One can also build a shared library from intermediate object files. The base shared library *libc.so.1* is a special case. It is normally built using the **-BSYMBOLIC=<list>** command line option of **ld**. Any call to a function whose name is on the list is bound at link time to the library's internal version of that function. Functions that are not on the list must be made available for replacement by a function with the same name in the application. (See the **ld** manual page for more information about its **-BSYMBOLIC** option.)

The new **cc** command line option **-BSYMBOLIC=<file>** supports building an intermediate version of *libc.so.1*. Each line the **<file>** names a function that can be put inline if it meets the inline substitution limits that are in effect. A function whose name is not on the list can not be put inline. The same file (NBSYMBS) that is used to build the **<list>** for **ld** can serve as the **<file>** for this new **cc** command line option. The comment lines (that start with the **#** character) and the trailing commas in NBSYMBS are ignored.

A function name can appear in the control files for both the **-BSYMBOLIC=<file>** and the **-F=<file>** options. The effect is to override the calls, stack size, and node size limits for functions that appear in both files.

Building Normal and Intermediate Object Files in the Same Directory

The default name for the intermediate object file for source file *x.c* is *x.o*. You can alter the file suffix for an intermediate object file by using the `-o` option of the `cc` command line. The command line

```
cc -c -ilobj -o x.oil ... x.c
```

builds *x.oil* as the intermediate object file that corresponds to source file *x.c*.

A makefile like this

```
.SUFFIXES:      .oil .il
.c.oil:
    cc -c -ilobj $(CFLAGS) $<-o $@
.SOURCES = dirx/x.c diry/y.c dirz/z.c
OBJECTS = $(SOURCES:.c=.o)
ILOBJECTS = $(SOURCES:.c=.oil)
xyz:      $(OBJECTS)
    cc -o $@ $(OBJECTS)
xyz.il: $(ILOBJECTS)
    cc -o $@ -ilobj $(ILOBJECTS)
```

can be used to build both the normal object files for *xyz* and the intermediate object files for *xyz.il* in the same directory.

Environment Variables With the Intermediate Compilation Facility

There are two environment variables that can be used to good effect with the final step—the one that builds the executable—when using intermediate object files. One of them can be used with any compilation and the other is applicable only to intermediate compiles.

TMPPREFIX and TMPDIR

TMPDIR is a system wide environment variable that controls where applications put their temporary files. TMPPREFIX is a compiler specific environment variable that serves this purpose specifically for the compiler. The compiler looks first for TMPPREFIX and then for TMPDIR. The default directory is */var/tmp*.

Intermediate object files are not small. Typically, an intermediate object file is up to six times the size of the corresponding normal object file. The combined size of the intermediate files that are presented to the inliner at one time to support building an executable from a large number of intermediate object files can be quite large. It may exceed the space that is available in */var/tmp*. The value of the TMPPREFIX (or TMPDIR if it is present and TMPPREFIX is not present) environment specifies the directory to use for temporary files during the a compilation. The normal object code files that result from the compilation of the intermediate code are among the files that are put in this directory.

ILOPT_ARGS

You may want to alter the options that apply to the optimizer for the compilation all of the final temporary object code files. The internal form of the optimizer options that apply to an intermediate file are stored in that file. The string that is the value of the ILOPT_ARGS environment variable is appended to this stored string. Since the ILOPT_ARGS string is appended to the internal form of the compiler options for the optimizer, it must also be in the internal form. The command line

```
ILOPT_ARGS='-off SSA' cc -ilobj -o xyz.ssa dirx/x.o diry/x.o
dirz/z.o
```

turns off compiler optimization SSA when the optimizer is compiling *x.o*, *y.o*, and *z.o*. (Note that a space is required at the end of the ILOPT_ARGS string.)

The command

```
strings -a -100 dirx/x.o
```

displays the stored option string for file *dirx/x.o*. Examining this output helps you to determine the string that you might want to use with the ILOPT_ARGS environment variable.

Normally, the strings command displays ASCII strings that occur in an object file. An intermediate object file has neither a text nor a data section, so that the strings command displays little if any more than the stored options string.

Output From a Command Line Using -Hih and -Hit=64

The files with suffix **.oil** in the following example output are intermediate output files. The library *ilobj.a* contains files *cluster.oil*, *list.oil*, *readGraph.oil*, *readLink.oil*, and *symbol.oil*.

```
cc -o cluster.il -ilobj -Hih -His=512 -Hit=64 main.oil ilobj.a
Inliner status for file "01234.main.oil":
  main():not inlined because tree nodes (=79) exceeds max of 64
Inliner status for file "01234.ilobj.a_cluster.oil":
  clusterLines():not inlined because tree nodes (=261) exceeds
max of 64
Inliner status for file "01234.ilobj.a_list.oil":
  append(): inlined, defined because is exported
  prepend(): inlined, defined because is exported
  Remove():not inlined because tree nodes (=91) exceeds max of
64
Inliner status for file "01234.ilobj.a_readGraph.oil":
  readCallGraph():not inlined because tree nodes (=89) exceeds
max of 64
Inliner status for file "01234.ilobj.a_readLink.oil":
  readLinkList():not inlined because tree nodes (=129) exceeds
max of 64
Inliner status for file "01234.ilobj.a_symbol.oil":
  hashIndex(): inlined
  addSymbol():not inlined because tree nodes (=119) exceeds max
of 64
  findSymbol():not inlined because tree nodes (=72) exceeds max
of 64
```

The object file names shown in the Inliner status for file *1234.xxx.oil* lines in the above example are those that cc assigns to those files in the temporary directory in which they are placed prior to their use in building the executable *cluster.il* in the example. When the inliner processes intermediate output files from an archive library (like *1234.list.oil* in the example), it prepends their names with that of the archive. This is necessary in cases where several object modules in the executable would otherwise have the same name. This does not occur in this example.

The algorithm to provide a unique name for each temporary object file uses the process id of the command line and the directory and archive that the intermediate object file occupies, as follows:

- **dir/lib(name)** becomes *processID.dir_lib_name*
- **dir/name** becomes *processID.dir_name*
- **lib(name)** becomes *processID.lib_name*
- **name** becomes *processID.name*

Index

- #
- # 2-6
- #include 10-4

-
- __HighC__ predefined macro F-30
- _abs C-31, C-32, F-24
- _alloca D-4
- _ASM F-36, F-37, F-38, F-39
- _compare F-24, F-26
- _fill_char F-24, F-25
- _find_char F-24, F-25
- _inb F-34
- _Inline C-16, D-2, D-4
- _inline(e1, e2, . . . en) F-35
- _insb F-34
- _max(e1,e2,...)/_min(e1,e2,...) F-24
- _max/_min F-24
- _move F-24, F-26
- _move_right F-24
- _offsetof F-24, F-26, F-27
- _packed F-29
- _RARELY F-27, F-28
- _REVERSE_PARMS F-31
- _skip_char F-24, F-25
- _unpacked F-29
- _USUALLY F-27, F-28
- _usually/-rarely F-24

- 3
- 386 2-6, C-29
- 387 4-3

- 4
- 486 2-6, 4-20, 4-27, 4-35, E-2, E-3
- 486 C-6, C-18, C-24, C-28, C-29

- 5
- 586 2-6, C-24, E-3

- 6
- 686 2-6, C-24

- 8
- 80386 E-2
- 80387 coprocessor
 - description 7-2
 - in-line instructions 7-3
- 80486 option
 - 586 C-24
 - 686 C-24

- A
- A 2-7

ABI_Args_in_regs 4-3
 aggregate object O F-5
 Aggressive_cloning 4-3, C-26
 Alias 3-3, 3-18
aliasing convention 3-18
 Align_all_labels 4-4, C-30
 Align_labels 4-4, C-30
 Align_members 3-3, C-30
 Align_routines 4-4, C-30
 alignment of structure members 3-13
 annotated source listing B-2, B-5, B-9
 ANSI preprocessor numbers F-41
 ar(1) I-3
 ARGS_IN_REGS C-18, C-29
 Args_in_regs_for_locals 4-5, C-28, C-29
 Arithmetic-simplification 4-5
 Arrays and Pointers 5-8
 Asm 4-6
 ASM facility E-6, F-36
 assembly routines 9-2
 assert statement E-5
 assignment for variables
 _ASM F-36
 attribute
 _REVERSE_PARMS F-31
 ARGS_IN_REGS C-29
 Auto_class_member_instantiation 4-6
 Auto_func_instantiation 4-7
 AutoDebug 4-7
 automatic aggregates F-14

B

Back_substitute_epilog 4-8, C-13
 basic block execution reporting
 profcnt(1) C-33, C-35
 basic block optimization

 option
 path=file_pathname C-26
 option
 dir=directory_name C-26
 merge=y C-27
 pid=y C-26
 basic block optimizing C-25
 basic block profiling C-26, C-33, C-35
 -qb C-35
 -qs C-33, C-35
 bbarc.out C-25, C-27
 bbarc.out file H-1
 BBPROFOPTS C-26
 Behaved 4-8, C-19
 Bit Fields 5-12
 Borland 4-8
 -BSYMBOLIC I-7
 -BSYMBOLIC I-7
 byte placement in code
 _inline(e1, e2, ... en) F-35
 bytes, in-line placement F-35

C

-c C-32, C-35
 -C 2-7, C-5, C-6, C-33, C-35, C-37, H-2
 c C-3
 C language
 description 1-2
 X3J11 extensions F-3
 call count information
 -F=file C-33
 -zcallgraph=file C-33
 call graph
 profcnt H-2
 Call Graph Utility H-1
 Called_infrequently 3-3

Callee_pops_when_possible 4-9,
 C-28, C-29
 caller callee I-5
 calling convention
 _CALLEE_POPS_STACK 8-3
 _RETURN_AGGREGATE_AS_
 POINTER 8-3
 _REVERSE_PARAMS 8-3
 NAME 8-3
 Calling_convention 3-4, C-28, C-29
 Calling_convention pragma 8-2
 calling-convention attribute F-31
 case ranges F-14
 case shifting external names in
 aliasing conventions 3-20
 cc command 2-2
 cgsort H-4
 changing default code generation
 strategy C-24
 Char_default_unsigned 4-9
 Char_is_rep 4-10
 Characters 5-3
 Cleanup_spills 4-10, C-16
 Cmd_line_ident 4-10
 code, in-line placement of bytes F-35
 command line option
 -c H-2
 -ilobj I-3
 -q**b** H-1
 -q**s** H-1
 command line options 2-6
 comments, C++ F-14
 communication with other
 languages F-23
 compiler intrinsics C-31
 Compiler Limits 5-19
 Compiler Maps Variables 5-10

compiler option -Hefmat — set format
 of error messages A-22
 compiler option -Hpragma — invoke
 a pragma on the
 commandline 3-2, A-25
 compiler option -Hwformat — set
 format of warningmessages A-22
 compiler option -w — set thecompiler
 diagnostic warninglevel A-23
 compiler toggles 4-1, 4-3
 constants, multi-character F-7
 controlling individual warning
 messagesbynumber A-23
 controlling warning messages A-23
 coprocessor, 80387 7-2
 cross references
 features B-2
 format B-7
 Cross_jump 4-11, C-11
 cross-reference format
 annotated source listing B-7
 file table B-7
 module name list B-7
 object-name cross reference B-7
 cross-referencer pragmas B-5
 customizing the compiler to a local
 environment 3-2

D

data communication 9-8
 Data pragma 8-7
 data segmentation 8-7
 declarators
 maximum number 5-18
 defaults file 2-24
 Define_static_members 4-11
 Demote_sizes 4-11
 dir=directory_name C-26

- direct I/O to hardware ports F-34
- direct register assignment F-36, F-37
- direction bit, stack frame 6-4
- Dname 2-7
- Double_return 4-12
- dump(1) I-3
- dumpcnt(1) C-25

E

- E 2-7
- Emit_names 4-12
- Empty_is_void 4-12
- enabling and disabling compiler
 - warning messages by
 - number 3-9, A-23
- Enforce_access_control 4-13
- enhanced ASM facility E-6
- Ensure_instantiation 3-5, 4-13
- enum, short/long F-40
- Enumerations 5-12
- environment variables I-9
- environment, identifying 2-5
- Epilog_call 3-5
- Epilog_trace 4-13
- epilogues 6-1, 6-5
- error and warning messages A-8
- errors
 - constraint A-5
 - file I/O A-2
 - lexical A-4
 - syntactic A-5
 - system A-3
 - user A-4
- escape sequences F-4
- examples of name unmangling 11-7
- Exception_aware_class 4-14
- Exclude_instantiation 3-5
- Exit_expr 3-5

- Export_vtabs 4-14
- extensions, High C F-13
- extern functions
 - duplicate declarations 8-4
- external name clashes and linker
 - limitations 3-18

F

- F 2-7
- F=<file> I-5, I-6, I-7
- F=file C-33, C-38
- Fast_virtual_bases 4-14
- fdouble 2-7
- file
 - hc.cnf C-17
- file I/O errors A-2
- file names, distinction of B-10
- file table B-8, B-9
- floating-point constants
 - suffixes f and l F-4
- floating-point divide operation C-31
- floating-point multiplication C-31
- Floating-Point Numbers 5-7
- floating-point processor 7-2
- for loop F-15, F-42, F-43, F-44, F-45,
 - F-46, F-47, F-49, F-50, F-51,
 - F-52, F-54, F-55, F-56
- format strings 2-26
- frame C-6, C-28
- frame pointer C-18
- frame removal C-17
- fsingle 2-7
- fstrict 2-7
- full function variables F-19
- function
 - global 9-3
- function calling information
 - qs I-5

function reordering H-3

functions

 naming conventions 9-3

 results 6-8

Fwait_before_return 4-16

G

-g 2-7, 4-38

getFieldUse H-8

global identifiers 3-3

Global_aliasing_convention 3-6, 3-19

Global_CSE 4-16, C-13

Globals_volatile 4-16

H

-H 2-7

-H except 2-8

-Hanno 2-8

-Hansi 2-8

hardware ports, direct I/O to F-34

-Hasm 2-8

-Hbbarc 2-8, 4-4, 4-22, C-25, C-26

Hbbarc=filename C-25

hc.cnf file C-16, C-17

-Hc_wrap 2-8

-Hcomdat 2-9

-Hcplus 2-9

-Hcpp 2-9

-Hcppext 2-11

hcxref B-1, B-3, B-6

hcxref command B-3

-Hefile 2-11

-Hefmat 2-12

-Hi 2-12, C-16, D-2

-Hia C-16, D-3, D-5

-Hic C-16, C-17, D-3, D-4, D-5, I-4,
I-5

-Hic=n C-16, D-3, D-5, H-2, I-4

High C extensions F-13

High_level_scheduling 4-16

-Hih C-37, D-3, D-5, D-6, I-1, I-4,
I-5, I-11

-Hin C-16, D-3, D-5

-Hinlsize 2-12

-Hir C-16, D-3, D-5

-His C-17, D-3, I-4, I-5, I-11

-His=<size> I-4

-His=n C-17, D-3, H-2

-Hit C-16, C-17, C-37, D-3, D-4, D-5,
I-1, I-4, I-5, I-11

-Hit=24, -Hic=24 C-16

-Hit=n D-3, D-5, H-3

-Hiu C-17, C-30

-Hiw C-17, C-30, D-3

-Hix C-17, D-3, D-5

-Hiz C-30, C-37, D-3, I-4

-Hkanji 2-12, F-33

-Hlines 2-12

-Hlist 2-13

-Hmake 2-13

-Hmakeof 2-13

-Hnocomdat 2-13

-Hnocplus 2-14

-Hnocpp 2-14, 10-4

-Hnoobject 2-14

-Hnopro 2-14

-Hon 2-14

-Hon=386_only C-24

-Hon=486_only C-24

-Hon=ncr_func_split H-4

-Hon=ncr_superblock C-26

-Hon=opt_input H-8

-Hon=p6_only C-24

-Hon=pentium_only C-24

-Hpcc 2-14

- Hpragma 2-15
- Hpro 2-14
- Hrel 2-15
- Hsrc 2-15
- Hsuffix 2-15
- Htmpprefix 2-15
- Humake 2-15
- Hunroll 2-15, C-21
- Hwide 2-15, C-30, D-3

I

- I/O operations to prot
 - option
 - _inb F-34
- I/O to hardware port
 - option
 - _insb F-34
- I/O to hardware ports F-34
- i386 ABI 8-2, F-29, F-32, F-40
- Ident 3-6
- Identifiers: 5-2
- identifying the environment 2-5
- Idir 2-16
- ilobj 2-16, C-37, I-3, I-4, I-5, I-8, I-10, I-11
- ILOPT_ARGS I-10
- Implementation Defined
 - Behavior" 5-2
- Import_vtabs 4-17
- Improved_spill_analysis 4-17
- Include file 10-3, 10-4
- Induction_analysis 4-17
- Init_stack 4-17
- initialization of data structures 3-16
- Initialization_level 3-6
- in-line assembly code 9-10
- in-line placement of bytes of
 - code F-35

- inline substitution I-3, I-4, I-5, I-7
 - controlling I-4
- inline substitution module I-1
 - command line option
 - F=<file> I-5
 - His=n H-2
 - Hit=n H-3
- inline substitution modular
 - Hic=n H-2
- Inline_common 4-18
- inliner
 - command line option
 - Hi D-2
 - definition D-2
 - option
 - Hia D-3
 - Hic=n D-3
 - Hih D-3
 - Hin D-3
 - Hir D-3
 - His D-3
 - Hit=n D-3
 - Hiw D-3
 - Hix D-3
 - Hiz D-3
 - option-His=n D-3
 - options D-3
- inlining
 - frame removal C-17
 - Hia C-16
 - Hic C-17
 - Hic=n C-16
 - Hin C-16
 - Hir C-16
 - His C-17
 - His=n C-17
 - Hit C-17
 - Hit=24, -Hic=24 C-16

- Hiu C-17
- Hiw C-17
- Hix C-17
- O C-17
- inlining C++ functions 3-8
- instantiating a function
 - template 11-26
- Int_function_warnings 4-18
- Integers 5-4
- Intel 80486 E-2
- intermediate compilation option
 - ilobj C-37
- Intermediate Compilation C-37, I-1, I-9
- intermediate object code files I-3
- intermediate object file
 - nm I-3
- intermediate object files
 - building libraries I-7
- intermediated object file
 - dump(1) I-3
- Interprocedural_optimizations 4-19
- intrinsic functions F-24
- intrinsics C-31
 - _abs C-32, F-24
 - _alloca D-4
 - _compare F-24
 - _fill_char F-24
 - _find_char F-24
 - _max/_min F-24
 - _move F-24
 - _move_right F-24
 - _offsetof F-24
 - _RARELY F-27
 - _skip_char F-24
 - _USUALLY F-27
 - _usually/-rarely F-24
- invoking

- C Macro Preprocessor 2-4
- High C compiler 2-2
- iterator F-42, F-43, F-44, F-45, F-47, F-48, F-49, F-50, F-51, F-52, F-53, F-54, F-55, F-56
- iterator, replacing with macros F-47
- iterator, use of F-43

K

- K 2-16
- Kanji
 - option
 - Hkanji F-33
- Kanji support F-33
- Keyword register 5-10

L

- language extensions for High C F-2
- languages other than High C 8-2
- law of associativity C-31
- lbbarc C-25
- ld C-33, C-37, C-38, H-3, H-4
 - command line option
 - BSYMBOLIC I-7
- ld(1) C-37, H-3
- line continuation character (\) F-3
- linker's external symbol
 - dictionary 3-18
- List 4-19
- listings, format of 10-3
- Literals_in_code 4-19
- Live_dead_iterate 4-19, C-12
- Local_CSE_iterate 4-20, C-12
- long double F-5
- Long_enums 4-20, F-40
- loop unrolling
 - description G-2

- requirements for loops G-4
- selecting loops for G-3
- Loop_factor 3-7, C-21, G-3, G-4
 - option
 - Hunroll C-21
- Loop_reg_rename 4-20, C-12
- Low_level_scheduling 4-20, C-14, C-15
- Low_level_scheduling_pass2 C-18

M

- macros
 - __HighC__predefined F-30
 - stringizing F-8
 - token pasting F-8
- Make_externs_global 4-21
- Mem_refs_from_loop 4-21, C-12
- merge=y C-27
- mergecnt(1) C-25
- module name list B-5, B-7, B-8, B-9
- mov instruction E-2
- multi-character constants F-7
- multi-module inlining
 - option
 - Hiu C-30
 - Hiw C-30
 - Hiz C-30
 - Hwide C-30

N

- name mangling 11-6
- named parameter association F-16
- NCR_predict_order 4-21
- NCRsetcc 4-21, C-14
- NCRsuperblock 4-21
- NCRswitch 4-22
- nested functions F-18

- Nested_types 4-22
- New_call_frame C-18
- nm I-3
- Noalias 4-24

O

- o 2-3, 2-18, C-3, C-17
- object-name cross reference B-7, B-8, B-9
- Oc C-23
- Off 3-7
- Off_inline 3-7
- Offwarn 3-7, A-23
- On 3-8
- On_inline 3-8
- Onwarn 3-8
- Onwarn_error 3-9
- optimization C-2, C-3, C-5, C-7, C-9, C-10, C-11, C-12, C-13, C-16, C-17, C-20, C-21, C-22, C-25, C-26, C-28, C-31
 - basic block C-25
 - cache
 - Oc C-23
 - executions speed vs. code
 - size C-5
 - level C-1, C-2, C-3
 - level 0 C-9
 - level 1 C-11
 - level 2 C-12
 - level 3 C-15
 - level 5 C-20
 - level 6 C-21
 - level s C-22
- optimization level
 - 0 C-3
 - 1-7 C-3
 - c C-3

- s C-3
- Optimize_for_space 4-24, C-22
- Optimize_FP 4-24, C-28
- optimized code
 - debugging C-4
- Overload_info 4-24
- overloading multiple functions with
 - the same name 11-6
- overriding default compiler
 - values 3-2

P

- P 2-18, I-5
- Pack 3-9
- Pack_text 4-25
- packed structures F-29
- Page 3-9
- page pragma 10-2
- parameter 6-2, 6-3, 6-8
- Parm_warnings 4-25
- path=file_pathname C-26
- PCC 4-25
- Pentium E-1, E-3
- Pentium option
 - 586 C-24
 - 686 C-24
- Pentium Pro C-24
- Pentium Pro option
 - 686 C-24
 - Hon=p6_only C-24
- pg 2-18
- pid=y C-26, C-27
- Pointers_compatible 4-26
- Pointers_compatible_with_ints 4-27
- Pop 3-9
- Pop_align_members 3-9
- Pop_ignore 3-9
- Pop_inline 3-10

- Popwarn 3-10
- portability 1-4
- Postfix_takes_two_args 4-27
- Postpone_arg_pops 4-27, C-29
- pragma 8-2, 8-5, 8-6, 8-7, 8-8, 9-8,
 - F-14, F-29, F-31, F-32
 - Alias 3-3, 3-18, 8-5, 9-3
 - Align_members 3-3, C-30
 - Align_members 3-13
 - Called_infrequently 3-3
 - calling convention 8-2
 - Calling_convention 3-4, C-28
 - Data 8-7, 9-8
 - Ensure_instantiation 3-5
 - Epilog_call 3-5
 - Exclude_instantiation 3-5
 - Exit_expr 3-5, 3-16, 3-17
 - Global_aliasing_convention 3-6,
 - 3-19
 - Ident 3-6, 3-20
 - Initialization_level 3-6, 3-16
 - Loop_factor 3-7, C-21, G-3
 - Off 3-7
 - Off_inline 3-7
 - Offwarn 3-7
 - On 3-8
 - On_inline 3-8
 - Onwarn 3-8
 - Onwarn_error 3-9
 - Pack 3-9, 3-14
 - Page 3-9, 10-2
 - Pop 3-9
 - Pop_align_members 3-9, 3-16
 - Pop_ignore 3-9
 - Pop_inline 3-10
 - Popwarn 3-10
 - Prolog_call 3-10
 - Push_align_members 3-10, 3-14

- Push_ignore 3-11
- Skip 3-11, 10-2
- Startup_expr 3-11, 3-16, 3-17
- Title 3-12, 10-2
- Warning_level 3-12
- Weak 3-12
- pragma as reserved word F-14
- Pragma Directives
 - behavior on encountering 5-19
- pragma Offwarn — disable turnoff
 - specified warningmessages A-24
- pragma Onwarn — enable turnon
 - specified warningmessages A-24
- pragma Onwarn_error — enable
 - specified warningmessages as compiler errormessages A-24
- pragma Warning_level — set compiler
 - diagnosticwarninglevel A-23
- Pragma_names B-5
- pragmas
 - aliasing 8-5
 - cross referencer B-5
- predicates, preprocessor E-5
- Preload_args_from_memory 4-28, C-11
- Preprocessing Directives 5-18
- preprocessor predicates E-5
- Print_lattice_members 4-28
- Print_lattices 4-28
- Print_layout 4-30
- Print_protos 4-30
- Print_template_usage 4-30
- Print_var_info 4-30
- processor-specific instructions
 - 386 C-29
 - 486 C-29
- prof(1) and gprof(1)

- command line option
 - p I-5
- profcnt H-2, H-6, H-7
- profcnt option
 - s C-35, C-36
- profcnt option-c C-33
- profcnt report types H-7
- profcnt(1) C-25, C-33, C-35
- profiles 2-23
- Prolog_call 3-10
- prologues 6-1, 6-5
- Prototype_conversion_warn 4-31
- Prototype_override_warnings 4-31
- prototypes F-57
- push instruction E-2
- Push_align_members 3-10
- Push_ignore 3-11

Q

- qb 4-22, C-25, C-26, C-35, H-1
- Qc 2-19
- qp 2-18
- qs C-33, C-35, C-38, H-1, H-2, H-4, H-6, H-8, I-5

R

- Read_only_strings 4-33, C-29
- Recognize_library 4-33
- Recognize_setjmp_longjmp 4-33
- Reduce_reg_contention 4-34, C-13
- Reg_alloc_enhance 4-34, C-13
- registers
 - nonvolatile 6-4
 - volatile 6-3
- related documentation xiv
- release definition xiii
- Reorder_scratch_regs 4-34

- reordering functions
 - option
 - Hon=ncr_func_split H-4
- Resolve_branch_targets 4-35
- resolving interfile dependencies
 - ld(1) I-3
- RTTI_comdat 4-35
- Run_time_type_info 4-35

S

- s C-3, C-22
- S 2-20, C-1, C-5, C-6, C-9, C-22,
 - C-35, C-36, C-37
- scalar argument 6-3
- scope 8-5, 8-8
- setting compiler pragmas 3-2
- signed F-5
- Skip 3-11
- skip pragma 10-2
- Source_code_order 4-36
- stack frame 6-2, 6-3, 6-5, 6-6, 6-7
 - direction bit 6-4
 - layout 6-2
- Startup_expr 3-11
- statement, C++ F-15
- static link 6-3, 6-5, 6-6
- static link method F-18
- Storage Mapping
 - for C++ class objects 5-1, 5-13,
 - 5-21
- Strength_reduce 4-36, C-20
- Strict_ANSI_math 4-36
- string constants F-3
- string constants, juxtaposed F-3
- stringizing F-8
- struct 10-3
- struct/enum F-5

- structure F-5, F-26, F-29, F-43, F-45,
 - F-46, F-52, F-54, F-56
- Structure Report H-1
- structure usage reporting H-6
- structured namespace unavailable in
 - C 3-18
- Structures 5-12
- suffixes u and l F-4
- switch statement
 - maximum case values 5-18

T

- Tail_recursion 4-36, C-28
- Template_trivial_conversions 4-36
- Title 3-12
- title pragma 10-2
- toggle
 - 387 4-3
 - 486 4-35
 - 486 C-29
 - ABI_Args_in_regs 4-3
 - Aggressive_cloning 4-3
 - Align_all_labels 4-4, C-30
 - Align_labels 4-4, C-30
 - Align_routines 4-4, C-30
 - Args_in_regs_for_locals 4-5,
 - C-28
 - Arithmetic_simplification 4-5
 - Asm 4-6
 - Auto_class_member_instantiation
 - 4-6
 - Auto_func_instantiation 4-7
 - AutoDebug 4-7
 - Back_substitute_epilog 4-8, C-13
 - Behaved 4-8, C-19
 - Borland 4-8
 - Callee_pops_when_possible 4-9,
 - C-28

- Char_default_unsigned 4-9
- Char_is_rep 4-10
- Cleanup_spills 4-10, C-16
- Cmd_line_idnt 4-10
- Command_line_idnt 3-21
- Cross_jump 4-11, C-11
- Define_static_members 4-11
- Demote_sizes 4-11
- Double_return 4-12
- Emit_names 4-12
- Empty_is_void 4-12
- Enforce_access_control 4-13
- Ensure_instantiation 4-13
- Epilog_trace 4-13
- Exception_aware_class 4-14
- Export_vtabs 4-14
- Fast_virtual_bases 4-14
- Fwait_before_return 4-16
- Global_CSE 4-16, C-13
- Globals_volatile 4-16
- High_level_scheduling 4-16
- Hon=ncr_superblock C-26
- Import_vtabs 4-17
- Improved_spill_analysis 4-17
- Induction_analysis 4-17
- Init_stack 4-17
- Inline_common 4-18
- Int_function_warnings 4-18
- Interprocedural_optimizations 4-19
- List 4-19
- Literals_in_code 4-19
- Live_dead_iterate 4-19, C-12
- Local_CSE_iterate 4-20, C-12
- Long_enums 4-20, F-40
- Loop_reg_rename 4-20, C-12
- Low_level_scheduling 4-20, C-14, C-15
- Low_level_scheduling_pass2 C-18
- Make_externs_global 4-21
- Mem_refs_from_loop 4-21, C-12
- NCR_predict_order 4-21
- NCRsetcc 4-21, C-14
- NCRsuperblock 4-21
- NCRswitch 4-22
- Nested_types 4-22
- New_call_frame C-18
- Noalias 4-24
- Optimize_for_space 4-24, C-22
- Optimize_FP 4-24, C-28
- Overload_info 4-24
- Pack_text 4-25
- Parm_warnings 4-25
- PCC 4-25
- Pointers_compatible 4-26
- Pointers_compatible_with_ints 4-27
- Postfix_takes_two_args 4-27
- Postpone_arg_pops 4-27
- Preload_args_from_memory 4-28, C-11
- Print_lattice_members 4-28
- Print_lattices 4-28
- Print_layout 4-30
- Print_protos 4-30
- Print_template_usage 4-30
- Print_var_info 4-30
- Prototype_conversion_warn 4-31
- Prototype_override_warnings 4-31
- Read_only_strings 4-33, C-29
- Recognize_library 4-33
- Recognize_setjmp_longjmp 4-33
- Reduce_reg_contention 4-34, C-13

- Reg_alloc_enhance 4-34, C-13
- Reorder_scratch_regs 4-34
- Resolve_branch_targets 4-35
- RTTI_comdat 4-35
- Run_time_type_info 4-35
- Source_code_order 4-36
- Strength_reduce 4-36, C-20
- Strict_ANSI_math 4-36
- Tail_recursion 4-36, C-28
- Template_trivial_conversions 4-36
 - 6
- Treat_exports_as_locals 4-37
- Trigraphs 4-37
- Try_static_cast 4-38
- unordered_compares 4-38
- Use_BSS 4-38
- Use_frame_pointer 4-38, C-18, C-28
- Use_reciprocal_for_divide 4-39, C-30
- Use_UP_rules 4-39
- VP_UP_warn 4-40
- Vtable_common 4-40
- Widen_float_args 4-41
- toggle Msgno — output error
 - messagenumbers A-24
- toggle Template_trivial_conversions
 - allow trivial conversions for matching arguments to functiontemplates 11-27
- toggles
 - compiler 4-3
 - definition of 4-2
- token-pasting F-8
- Treat_exports_as_locals 4-37
- Trigraphs 4-37, F-11
- truncationinaliasing conventions 3-20
- Try_static_cast 4-38
- type modifier
 - long double F-5
 - signed F-5
- type qualifier
 - _Inline D-2
- type-safe linkage 11-6
- U
 - Uname 2-20
- unassert statement E-5
- undeclared functions 8-3, 8-4, A-25
- union 10-3
- Unions 5-12
- unmangle — nameunmangler converts
 - mangled function and variablenames 11-7
- unordered_compares 4-38
- unpacked structures F-29
- up-level referencing F-18
- Use_BSS 4-38
- Use_frame_pointer 4-38, C-18, C-28
- Use_reciprocal_for_divide 4-39, C-30, C-31
- Use_UP_rules 4-39
- user errors A-4
- using unmangle in debugging 11-7
- V
 - V 2-20
- vertical tab F-3, F-4
- vertical tab as delimiter F-3
- volatile object
 - accessing 5-17
- VP_UP_warn 4-40
- Vtable_common 4-40

W

-W 2-20

warning levels A-7

warning messages A-8

Warning_level 3-12

Warning_level pragma A-7

Weak 3-12

Widen_float_args 4-41

-wn 2-20

X

X3J11 extensions to C F-3

-Xa 2-20, 2-21

-Xc 2-20, 2-21

-Xt 2-20, 2-21

Y

-YI 2-21

-YP 2-21

-YS 2-21

Z

-zcallgraph=file C-33

-Zp 2-22