C++ ユーザーズガイド

Sun™ Studio 8

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-5807-10
2004 年 3 月 , Revision A

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者が開発したソフトウェアが含まれている可能性があります。

フォント技術を含む第三者のソフトウェアは、著作権法により保護されており、提供者からライセンスを受けているものです。

libdwarf and lidredblack are Copyright 2000 Silicon Graphics Inc. and are available under the GNU Lesser General Public License from http://www.sgi.com.

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。

Sun、Sun Microsystems、Forte、Java、iPlanet、NetBeans および docs.sun.com は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

サンのロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。

Netscape および Netscape Navigator は、米国ならびに他の国における Netscape Communications Corporation の 商標または登録商標です。

すべての SPARC の商標はライセンス規定に従って使用されており、米国および他の各国における SPARC International, Inc. の商標または登録商標です。SPARC の商標を持つ製品は、Sun Microsystems, Inc. によって開発されたアーキテクチャに基づいています。

このマニュアルに記載されている製品および情報は、米国の輸出規制に関する法規の適用および管理下にあり、また、米国以外の国の輸出および輸入規制に関する法規の制限を受ける場合があります。核、ミサイル、生物化学兵器もしくは原子力船に関連した使用またはかかる使用者への提供は、直接的にも間接的にも、禁止されています。このソフトウェアを、米国の輸出禁止国へ輸出または再輸出すること、および米国輸出制限対象リスト(輸出が禁止されている個人リスト、特別に指定された国籍者リストを含む)に指定された、法人、または団体に輸出または再輸出することは一切禁止されています。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含み、明示的であるか黙示的であるかを問わず、あらゆる説明および保証は、法的に無効である限り、拒否されるものとします。

原典： C++ User's Guide
Part No: 817-5070-10
Revision A

© 2004 by Sun Microsystems, Inc.

Adobe PostScript

目次

はじめに　xxvii

Part I　C++ コンパイラ

1. C++ コンパイラの紹介　1

標準の準拠　1

C++ README ファイル　2

マニュアルページ　2

ライセンス　3

C++ コンパイラの新機能　3

全般的な強化機能　3

コンパイルの高速化　6

移植の簡略化　9

パフォーマンスの向上　9

警告とエラーの新しい制御機能　11

C++ ユーティリティ　12

各国語のサポート　13

2. C++ コンパイラの使用方法　15

コンパイル方法の概要　15

コンパイラの起動　17
コマンド構文　17
ファイル名に関する規則　18
複数のソースファイルの使用　19
バージョンが異なるコンパイラでのコンパイル　19
コンパイルとリンク　21
コンパイルとリンクの流れ　21
コンパイルとリンクの分離　21
コンパイルとリンクの整合性　22
SPARC V9 のためのコンパイル　23
コンパイラの診断　23
コンパイラの構成　25
指示および名前の前処理　26
プラグマ　26
可変数の引数をとるマクロ　26
事前に定義されている名前　27
#error　27
メモリー条件　27
スワップ領域のサイズ　28
スワップ領域の増加　28
仮想メモリーの制御　28
メモリー条件　29
コマンドの簡略化　30
C シェルでの別名の使用　30
CCFLAGS によるコンパイルオプションの指定　30
make の使用　31
3. C++ コンパイラオプションの使い方　33

構文　33
一般的な注意事項　33
機能別に見たオプションの要約　34
コード生成オプション　35
コンパイル時パフォーマンスオプション　36
デバッグオプション　36
浮動小数点オプション　38
言語オプション　38
ライブラリオプション　39
ライセンスオプション　40
廃止オプション　40
出力オプション　41
実行時パフォーマンスオプション　42
プリプロセッサオプション　44
プロファイルオプション　45
リファレンスオプション　45
ソースオプション　45
テンプレートオプション　46
スレッドオプション　46

Part II　C++ プログラムの作成

4. 言語拡張　49

リンカースコープ　50
スレッドローカルな記憶装置　51
例外の制限の少ない仮想関数による置き換え　52
enum 型と enum 変数の前方宣言　52
不完全な enum 型の使用　53

enum 名のスコープ修飾子としての使用　54
名前のない struct 宣言の使用　54
名前のないクラスインスタンスのアドレスの受け渡し　56
静的名前空間スコープ関数のクラスフレンドとしての宣言　57
事前定義済み __func__ シンボルの関数名としての使用　58

5. プログラムの編成　59

ヘッダーファイル　59
言語に対応したヘッダーファイル　59
べき等ヘッダーファイル　61
テンプレート定義　61
テンプレート定義の取り込み　62
テンプレート定義の分離　63

6. テンプレートの作成と使用　65

関数テンプレート　65
関数テンプレートの宣言　65
関数テンプレートの定義　66
関数テンプレートの使用　66
クラステンプレート　67
クラステンプレートの宣言　67
クラステンプレートの定義　68
クラステンプレートメンバーの定義　68
クラステンプレートの使用　70
テンプレートのインスタンス化　70
テンプレートの暗黙的インタンス化　71
テンプレートの明示的インスタンス化　71
テンプレートの編成　72

デフォルトのテンプレートパラメータ　73
テンプレートの特殊化　73
テンプレートの特殊化宣言　74
テンプレートの特殊化定義　74
テンプレートの特殊化の使用とインスタンス化　75
部分特殊化　75
テンプレートの問題　76
非局所型名前の解決とインスタンス化　76
テンプレート引数としての局所型　78
テンプレート関数のフレンド宣言　78
テンプレート定義内での修飾名の使用　81
テンプレート宣言の入れ子　81
静的変数や静的関数の参照　82
テンプレートを使用して複数のプログラムを同一ディレクトリに構築する　82

7. テンプレートのコンパイル　85

冗長コンパイル　85
テンプレートのインスタンス化　86
生成されるインスタンス　86
全クラスインスタンス化　86
コンパイル時のインスタンス化　87
テンプレートインスタンスの配置とリンケージ　87
外部インスタンス　88
静的インスタンス　90
大域インスタンス　91
明示的インスタンス　91
半明示的インスタンス　92
テンプレートリポジトリ　92

リポジトリの構造　92
テンプレートリポジトリへの書き込み　93
複数のテンプレートリポジトリからの読み取り　93
テンプレートリポジトリの共有　93
-instance=extern による テンプレートインスタンスの自動一貫性　94
テンプレート定義の検索　94
ソースファイルの位置規約　94
定義検索パス　95
テンプレートオプションファイル　95
注釈　96
インクルード　96
ソースファイルの拡張子　96
定義ソースの位置　97
テンプレートの特殊化エントリ　99

8. 例外処理　103

同期例外と非同期例外　103
実行時エラーの指定　103
例外の無効化　104
実行時関数と事前定義済み例外の使用　105
シグナルや Setjmp/Longjmp と例外との併用　106
例外のある共有ライブラリの構築　107

9. キャスト演算　109

cast キャスト　110
解釈を変更するキャスト　110
静的キャスト　112
動的キャスト　112

階層の上位にキャストする　113
void* にキャストする　113
階層の下位または全体にキャストする　113

10. プログラムパフォーマンスの改善　119

一時オブジェクトの回避　119
インライン関数の使用　120
デフォルト演算子の使用　121
値クラスの使用　122
クラスを直接渡す　123
各種のプロセッサでクラスを直接渡す　124
メンバー変数のキャッシュ　124

11. マルチスレッドプログラムの構築　127

マルチスレッドプログラムの構築　127
マルチスレッドコンパイルの確認　128
C++ サポートライブラリの使用　128
マルチスレッドプログラムでの例外の使用　129
C++ 標準ライブラリのオブジェクトのスレッド間での共有　129
マルチスレッド環境での従来の iostream の使用　133
マルチスレッドで使用しても安全な iostream ライブラリの構成　133
iostream ライブラリのインタフェースの変更　141
大域データと静的データ　143
連続実行　144
オブジェクトのロック　145
マルチスレッドで使用しても安全なクラス　147
オブジェクトの破棄　148
アプリケーションの例　149

Part III　ライブラリ

12. ライブラリの使用　155

C ライブラリ　155

C++ コンパイラ付属のライブラリ　156

C++ライブラリの説明　157

C++ ライブラリのマニュアルページへのアクセス　159

デフォルトの C++ ライブラリ　160

関連するライブラリオプション　160

クラスライブラリの使用　162

`iostream` ライブラリ　162

`complex` ライブラリ　164

C++ライブラリのリンク　166

標準ライブラリの静的リンク　166

共有ライブラリの使用　167

C++ 標準ライブラリの置き換え　169

置き換え可能な対象　170

置き換え不可能な対象　170

代替ライブラリのインストール　170

代替ライブラリの使用　171

標準ヘッダーの実装　171

13. C++ 標準ライブラリの使用　175

C++ 標準ライブラリのヘッダーファイル　176

C++ 標準ライブラリのマニュアルページ　178

STLport　193

14. 従来の `iostream` ライブラリの使用　195

定義済みの iostream 196
iostream 操作の基本構造 196
従来型の iostream ライブラリの使用 197
iostream を使用した出力 198
iostream を使用した入力 202
ユーザー定義の抽出演算子 202
char*の抽出子 203
1 文字の読み込み 204
バイナリ入力 204
入力データの先読み 205
空白の抽出 205
入力エラーの処理 205
iostream と stdio の併用 206
iostreamの作成 207
クラス fstream を使用したファイル操作 207
iostream の代入 211
フォーマットの制御 212
マニピュレータ 212
引数なしのマニピュレータの使用法 214
引数付きのマニピュレータの使用法 215
Strstreams：配列用の iostreams 217
Stdiobufs ： 標準入出力ファイル 用の iostream 217
Streambufs 217
streambuf の機能 218
streambuf の使用 218
iostream に関するマニュアルページ 219
iostream の用語 222

15. 複素数演算ライブラリの使用　225

複素数ライブラリ　225

複素数ライブラリの使用方法　226

complex 型　226

complex クラスのコンストラクタ　226

算術演算子　227

数学関数　228

エラー処理　230

入出力　231

混合演算　232

効率　233

複素数のマニュアルページ　234

16. ライブラリの構築　235

ライブラリとは　235

静的 (アーカイブ) ライブラリの構築　236

動的 (共有) ライブラリの構築　237

例外を含む共有ライブラリの構築　238

非公開ライブラリの構築　239

公開ライブラリの構築　239

C API を持つライブラリの構築　240

dlopen を使って C プログラムから C++ ライブラリにアクセスする　241

Part IV　付録

A. C++ コンパイラオプション　245

オプション情報の構成　246

オプションの一覧　247

-386 247

-486 247

-a 247

-Bbinding 248

-c 250

-cg{89|92} 250

-compat[={4|5}] 250

+d 252

-D[ ]*name*[=*def*] 253

-d{y|n} 255

-dalign 256

-dryrun 257

-E 257

+e{0|1} 258

-erroff[=t] 259

-errtags[=a] 260

-errwarn[=t] 261

-fast 262

-features=*a*[,a...] 265

-filt[=filter[,filter...]] 270

-flags 273

-fnonstd 274

-fns[={yes|no}] 274

-fprecision=*p* 276

-fround=*r* 277

-fsimple[=*n*] 278

-fstore 280

-ftrap=*t*[,*t*...] 280

-G 282
-g 283
-g0 285
-H 285
-h[ ]name 285
-help 286
-Ipathname 286
-I- 287
-i 290
-inline 290
-instances=*a* 290
-instlib=<出力ファイル> 291
-KPIC 292
-Kpic 292
-keeptmp 293
-L*path* 293
-l*lib* 293
-libmieee 294
-libmil 294
-library=*l*[,*l*...] 294
-mc 300
-migration 300
-misalign 301
-mr[,string] 302
-mt 302
-native 303
-noex 303
-nofstore 303

-nolib 304
-nolibmil 304
-noqueue 304
-norunpath 304
-O 305
-Olevel 305
-o filename 305
+p 306
-P 306
-p 306
-pentium 307
-pg 307
-PIC 307
-pic 307
-pta 307
-ptipath 308
-pto 308
-ptr 308
-ptv 309
-Qoption phase option[,option...] 309
-qoption phase option 310
-qp 310
-Qproduce sourcetype 310
-qproduce sourcetype 310
-Rpathname[:pathname...] 311
-readme 311
-S 311
-s 312

-sb 312

-sbfast 312

-staticlib=l[,l...] 312

-temp=*path* 315

-template=opt[,opt...] 315

-time 317

-U *name* 317

-unroll=*n* 318

-V 318

-v 318

-vdelx 318

-verbose=v[,v...] 319

+w 320

+w2 321

-w 321

-Xm 322

-xa 322

-xalias_level[=n] 322

-xar 325

-xarch=*isa* 326

-xbuiltin[={%all|%none}] 332

-xcache=*c* 333

-xcg89 335

-xcg92 335

-xchar[=o] 336

-xcheck[=i] 337

-xchip=*c* 338

-xcode=*a* 340

-xcrossfile[=n] 342

-xdumpmacros[=value[,value...]] 344

-xe 349

-xF[=*v*[,*v*...]] 349

-xhelp=flags 351

-xhelp=readme 351

-xia 351

-xildoff 352

-xildon 352

-xinline[=func_spec[,func_spec...]] 353

-xipo[={0|1|2}] 355

-xjobs=*n* 358

-xlang=language[,language] 358

-xldscope={*v*} 360

-xlibmieee 362

-xlibmil 363

-xlibmopt 363

-xlic_lib=sunperf 364

-xlicinfo 365

-xlinkopt[=*level*] 365

-xM 367

-xM1 368

-xMerge 368

-xmemalign=*ab* 369

-xnativeconnect[=i] 370

-xnolib 372

-xnolibmil 374

-xnolibmopt 374

-xOlevel 375

-xopenmp[=i] 379

-xpagesize=*n* 380

-xpagesize_heap=*n* 381

-xpagesize_stack=*n* 382

-xpch=v 383

-xpchstop=file 387

-xpg 388

-xport64[=(*v*)] 389

-xprefetch[=a[,a...]] 393

-xprefetch_level[=i] 396

-xprofile=p 397

-xprofile_ircache[=path] 400

-xprofile_pathmap 401

-xregs=r[,r...] 402

-xs 404

-xsafe=mem 404

-xsb 405

-xsbfast 405

-xspace 406

-xtarget=t 406

-xthreadvar[=*o*] 414

-xtime 415

-xtrigraphs[={yes|no}] 415

-xunroll=n 417

-xustr={ascii_utf16_ushort|no} 417

-xvis[={yes|no}] 418

-xwe 419

-z[ ]arg 419

B. プラグマ 421

プラグマの書式 421

プラグマの詳細 422

#pragma align 423

#pragma does_not_read_global_data 424

#pragma does_not_return 425

#pragma does_not_write_global_data 425

#pragma dumpmacros 426

#pragma end_dumpmacros 427

#pragma fini 427

#pragma hdrstop 428

#pragma ident 428

#pragma init 428

#pragma no_side_effect 429

#pragma pack(n) 430

#pragma rarely_called 432

#pragma returns_new_memory 432

#pragma unknown_control_flow 433

#pragma weak 433

用語集 437

索引 447

表目次

表 2-1　C++ コンパイラが認識できるファイル名接尾辞　18
表 2-2　C++ コンパイルシステムの構成要素　25
表 3-1　オプション構文形式の例　33
表 3-2　コード生成オプション　35
表 3-3　コンパイル時パフォーマンスオプション　36
表 3-4　デバッグオプション　36
表 3-5　浮動小数点オプション　38
表 3-6　言語オプション　38
表 3-7　ライブラリオプション　39
表 3-8　ライセンスオプション　40
表 3-9　廃止オプション　40
表 3-10　出力オプション　41
表 3-11　実行時パフォーマンスオプション　42
表 3-12　プリプロセッサオプション　44
表 3-13　プロファイルオプション　45
表 3-14　リファレンスオプション　45
表 3-15　ソースオプション　45
表 3-16　テンプレートオプション　46
表 3-17　スレッドオプション　46
表 4-1　宣言指定子　50
表 10-1　アーキテクチャ別の構造体と共用体の渡し方　124

表 11-1　iostream の中核クラス　134
表 11-2　マルチスレッドで使用しても安全な、再入可能な公開関数　135
表 12-1　C++ コンパイラに添付されるライブラリ　156
表 12-2　C++ ライブラリにリンクするためのコンパイラオプション　166
表 12-3　ヘッダー検索の例　173
表 13-1　C++ 標準ライブラリのヘッダーファイル　176
表 13-2　C++ 標準ライブラリのマニュアルページ　178
表 14-1　iostream ルーチンのヘッダーファイル　197
表 14-2　iostream の定義済みマニピュレータ　213
表 14-3　iostream に関するマニュアルページの概要　220
表 14-4　iostream の用語　222
表 15-1　複素数ライブラリの関数　229
表 15-2　複素数の数学関数と三角関数　229
表 15-3　複素数ライブラリ関数のデフォルトエラー処理　231
表 15-4　complex 型のマニュアルページ　234
表 A-1　オプション構文形式の例　245
表 A-2　オプションの見出し　246
表 A-3　SPARC と IA 用の事前定義シンボル　254
表 A-4　-erroff の値　259
表 A-5　-errwarn の値　261
表 A-6　-fast の拡張　262
表 A-7　互換モードと標準モードでの -feature オプション　265
表 A-8　標準モードだけに使用できる -features オプション　268
表 A-9　互換モードだけに使用できる -features オプション　268
表 A-10　-filt の値　271
表 A-11　-fns の値　275
表 A-12　-fprecision の値　276
表 A-13　-fround の値　278
表 A-14　-fsimple の値　279
表 A-15　-ftrap の値　281
表 A-16　-instances の値　290

表 A-17　互換モードに使用できる -library オプション　295
表 A-18　標準モードに使用できる -library オプション　296
表 A-19　-Qoption の値　309
表 A-20　-Qproduce の値　310
表 A-21　-staticlib の値　313
表 A-22　-template の値　316
表 A-23　-verbose の値　319
表 A-24　SPARC プラットフォームでの -xarch の値　327
表 A-25　IA プラットフォームでの -xarch の値　331
表 A-26　-xcache の値　334
表 A-27　-xchar の値　336
表 A-28　-xcheck の値　337
表 A-29　-xchip の値　339
表 A-30　-xcode の値　340
表 A-31　-xcrossfile の値　343
表 A-32　-xdumpmacros の値　344
表 A-33　-xF の値　350
表 A-34　-xinline の値　353
表 A-35　-xipo の値　356
表 A-36　-xldscope の値　361
表 A-37　-xlinkopt の値　366
表 A-38　-xmemalign の境界整列と動作の値　369
表 A-39　-xmemalign の例　370
表 A-40　-xnativeconnect の値　371
表 A-41　-xopenmp の値　379
表 A-42　-xport64 の値　389
表 A-43　-xprefetch の値　393
表 A-44　-xprefecth_level の値　396
表 A-45　-xregs の値　403
表 A-46　SPARC プラットフォームの-xtargetの値　407
表 A-47　-xtarget の SPARC プラットフォーム名　407

表 A-48　IA プラットフォームの -xtargetの値　412
表 A-49　Intel アーキテクチャでの -xtarget の展開　412
表 A-50　-xthreadvar の値　414
表 A-51　-xtrigraphs の値　416
表 B-1　プラットフォームのもっとも厳密な境界整列　430
表 B-2　メモリーサイズとデフォルトの境界整列 (単位はバイト数)　431

コード例目次

コード例 6-1　テンプレート引数としての局所型の問題の例　78
コード例 6-2　フレンド宣言の問題の例　79
コード例 7-1　冗長な definition エントリ　97
コード例 7-2　静的なデータメンバーの定義と単純名の使用　98
コード例 7-3　テンプレートメンバー関数の定義　98
コード例 7-4　異なるソースファイルにあるテンプレート関数の定義　98
コード例 7-5　`nocheck` オプション　99
コード例 7-6　`special` エントリ　100
コード例 7-7　`special` エントリを使用する必要がある場合　100
コード例 7-8　`special` エントリの多重定義　101
コード例 7-9　テンプレートクラスの特殊化　101
コード例 7-10　静的テンプレートクラスメンバーの特殊化　101
コード例 11-1　エラー状態のチェック　137
コード例 11-2　`gcount` の呼び出し　138
コード例 11-3　ユーザー定義の入出力操作　138
コード例 11-4　マルチスレッドでの安全性の無効化　139
コード例 11-5　マルチスレッドでの安全性の無効化　140
コード例 11-6　マルチスレッドで使用すると安全ではないオブジェクトの同期処理　140
コード例 11-7　新しいクラス　141
コード例 11-8　新しいクラス階層　141
コード例 11-9　新しい関数　142

コード例 11-10　ロック処理の使用例　146
コード例 11-11　入出力操作とエラーチェックの不可分化　147
コード例 11-12　共有オブジェクトの破棄　148
コード例 11-13　iostreamオブジェクトをマルチスレッドで使用しても安全な方法で使用　149
コード例 14-1　string の抽出演算子　202
コード例 A-1　プルプロセッサのプログラム例 foo.cc　257
コード例 A-2　-E オプションを使用したときの foo.cc のプリプロセッサ出力　258

はじめに

このマニュアルでは、Sun™ Open Network Environment (Sun ONE) Studio 8, Compiler Collection の C++ コンパイラの使用方法を説明し、コマンド行コンパイラオプションに関する詳しい情報を提供します。このマニュアルは、C++ に関する実用的な知識と Solaris™ のオペレーティング環境と UNIX® コマンドに関する実用的な知識を持つプログラマを対象にしています。

内容の紹介

このマニュアルで取り上げるトピックは次のとおりです。

C++ コンパイラ: 第 1 章では標準の準拠や新機能など、コンパイラに関する初歩的な内容について説明します。第 2 章ではコンパイラの使用方法を説明し、第 3 章ではコンパイラのコマンド行オプションの使用方法を説明します。

C++ プログラムの作成方法: 第 4 章では、他の C++ コンパイラによって一般に受け入れられる非標準コードのコンパイル方法を説明します。第 5 章では、ヘッダーファイルやテンプレート定義の設定および構成について提案します。第 6 章では、テンプレートの作成方法や使用方法を説明します。第 7 章では、テンプレートをコンパイルする際の各種オプションについて説明します。例外処理については第 8 章、キャスト演算に関しては第 9 章で説明します。第 10 章では、Sun WorkShop C++ コンパイラに大きな影響を与えるパフォーマンス手法について説明します。第 11 章では、マルチスレッド化プログラムの構築に関する情報を提供します。

ライブラリ: 第 12 章では、コンパイラに組み込まれているライブラリの使用方法を説明します。C++ 標準ライブラリについては第 13 章、典型的な iostream ライブラリ (互換モードの場合) については第 14 章で説明し、複素数演算ライブラリ (互換モードの場合) については第 15 章で説明します。第 16 章ではライブラリの構築に関する情報を提供します。

コンパイラのオプション: 付録 A ではコンパイラのオプションについて詳しく説明します。

書体と記号について

次の表と記述は、このマニュアルで使用している書体と記号について説明しています。

書体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `machine_name% You have mail.`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	`machine_name%` **`su`** `Password:`
AaBbCc123 または **ゴシック**	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	`rm` *filename* と入力します。 `rm` **ファイル名** と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』

書体または記号	意味	例
「」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	`machinename% grep '^#define \` `  XV_VERSION_STRING'`
‰	階層メニューのサブメニューを選択することを示します。	作成: 「返信」‰「送信者へ」

コード記号	意味	表記	コード例
[ ]	角括弧は、省略可能な引数を示します。	`-compat[=n]`	`-compat=4`
{ }	中括弧は、必須オプションの選択肢を示します。	`d{y|n}`	`-dy`
|	パイプ記号 (縦線) は、複数の引数のうちいずれか 1 つだけを選択することを示します。	`B{dynamic|static}`	`-Bstatic`
:	コロンは、コンマと同様に、引数を区切る場合に使用します。	`R`*dir*`[:`*dir*`]`	`-R/local/libs:/U/a`
...	省略符号は、一連のオプションでの省略部分を示します。	`-xinline=`*f1*`[,...`*fn*`]`	`-xinline=alpha,dos`

シェルプロンプトについて

シェル	プロンプト
UNIX の C シェル	machine_name%
UNIX の Bourne シェルと Korn シェル	machine_name$
スーパーユーザー (シェルの種類を問わない)	#

コンパイラコレクションのツールとマニュアルページへのアクセス

コンパイラコレクションのコンポーネントとマニュアルページは、標準の /usr/bin/ と /usr/share/man の各ディレクトリにインストールされません。コンパイラとツールにアクセスするには、PATH 環境変数にコンパイラコレクションのコンポーネントディレクトリを必要とします。マニュアルページにアクセスするには、PATH 環境変数にコンパイラコレクションのマニュアルページディレクトリが必要です。

PATH 変数についての詳細は、csh(1)、sh(1) および ksh(1) のマニュアルページを参照してください。MANPATH 変数についての詳細は、man(1) のマニュアルページを参照してください。このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Sun ONE Studio コンパイラコレクションコンポーネントが /opt ディレクトリにインストールされていることを想定しています。ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

コンパイラとツールへのアクセス方法

PATH 環境変数を変更して、コンパイラとツールにアクセスできるようにする必要があるかどうか判断するには以下を実行します。

▼ PATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、PATH 変数の現在値を表示します。

```
% echo $PATH
```

2. 出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。

   パスがある場合は、PATH 変数はコンパイラとツールにアクセスできるように設定されています。パスがない場合は、次の指示に従って、PATH 環境変数を設定してください。

▼ PATH 環境変数を設定してコンパイラとツールにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

   /opt/SUNWspro/bin

マニュアルページへのアクセス方法

マニュアルページにアクセスするために MANPATH 変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ MANPATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、dbx マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

   dbx(1) マニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って MANPATH 環境変数を設定してください。

▼ MANPATH 変数を設定してマニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

   ```
   /opt/SUNWspro/man
   ```

コンパイラコレクションのマニュアルへのアクセス

マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。
  /opt/SUNWspro/docs/ja/index.html

製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、docs.sun.com の Web サイトで入手できます。以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (docs.sun.com Web サイトでは入手できません)。
  - 『Standard C++ Library Class Reference』
  - 『標準 C++ ライブラリ・ユーザーズガイド』
  - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
  - 『Tools.h++ ユーザーズガイド』

インターネットの docs.sun.com Web サイト (http://docs.sun.com) から、サンのマニュアルを参照したり、印刷したり、購入することができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 – Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル (サードパーティ製マニュアルは除く)	形式 : HTML (日本語版は PDF のみ) 場所 : http://docs.sun.com
サードパーティ製マニュアル: 『Standard C++ Library Class Reference』 『標準 C++ ライブラリ・ユーザーズガイド』 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 『Tools.h++ ユーザーズガイド』	形式 : HTML 場所 : file:/opt/SUNWspro/docs/ja/index.html のマニュアル索引
Readme および マニュアルページ	形式 : HTML 場所 : file:/opt/SUNWspro/docs/ja/index.html
リリースノート	製品 CD 内の HTML ファイル

関連するコンパイラコレクションのマニュアル

下表に、マニュアル索引 (file:/opt/SUNWspro/docs/ja/index.html) から利用できる関連マニュアルを示します。製品のソフトウェアが /opt ディレクトリにインストールされていない場合は、システム管理者に該当するパスを確認してください。

マニュアルのタイトル	説明
『数値計算ガイド』	浮動小数点数の計算精度についての問題を説明しています。

関連する Solaris マニュアル

次の表では、docs.sun.com の Web サイトで参照できる関連マニュアルについて説明します。

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris 8 Reference Manual Collection	マニュアルページの節を参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris 8 Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris 8 Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

C++ 関連マニュアルページ

本書では、C++ ライブラリで使用できるマニュアルページの一覧を提供します。次の表 には、それ以外の C++ に関連するマニュアルページを示します。

タイトル	内容
`c++filt`	ファイルを順番通りに読み、C++ の符号化された名前と思われるシンボルを復号化した後、標準出力に書き出す
`dem`	指定した複数の C++ 名の復号化
`fbe`	アセンブリ言語のソースファイルからオブジェクトファイルの作成
`fpversion`	システムの CPU と FPU に関する情報の出力

タイトル	内容
gprof	プログラムの実行プロファイルの作成
ild	プログラムの修正部分だけをリンクし、修正オブジェクトコードを以前に構築された実行可能ファイルに挿入することを可能にする
inline	インライン手続きの呼び出しの展開
lex	字句解析プログラムの生成
rpcgen	RPC プロトコルを実装するため C/C++ コードの生成
sigfpe	特定の SIGFPE コードに対するシグナル処理を許可
stdarg	変更可能な引数のリストを処理
varargs	変更可能な引数のリストを処理
version	オブジェクトファイルまたはバイナリファイルのバージョン識別情報の表示
yacc	文脈自由文法を、LALR(1) 構文解析アルゴリズムを実行する単純オートマトン用の一連の表に変換

市販の書籍

C++ について書かれている書籍の一部を紹介します。

『注解 C++ リファレンス・マニュアル』Margaret A. Ellis、Bjarne Stroustrup 共著、シイエム・シイ 2001 年

『 プログラミング言語 C++ 』第 3 版 Bjarne Stroustrup 著、アスキー、1997年

『C++ primer』改訂 3 版、Stanley B. Lippman、Josee Lajoie 共著、アスキー、2002 年

『Effective C++』改訂 2 版、Scott Meyers 著、アスキー、1998 年

『C++ 標準ライブラリチユートリアル & リファレンス』Nicolai Josuttis 著、アスキー、2001 年

『Generic Programming STL による汎用プログラミング』Matthew Austern 著、アスキー、2000 年

『Standard C++ IOStreams and Locales』Angelica Langer、Klaus Kreft 共著、Addison-Wesley、2000 年

『Thinking in C++』Volume 1、Second Edition、Bruce Eckel 著、Prentice Hall、1995 年

『オブジェクト指向における再利用のためのデザインパターン』
Erich Gamma、Richard Helm、Ralph Johnson,、John Vlissides 共著、ソフトバンクパブリッシング、1999 年

『More Effective C++ - 最新 35 のプログラミング技法』Scott Meyers 著、アスキー、1998 年

『Efficient C++ パフォーマンスプログラミングテクニック』Dov Bulka and David Mayhew 共著、ピアソンエデュケーション、2000 年

開発者向けのリソース

`http://www.sun.com/developers/studio` にアクセスし、Compiler Collection というリンクをクリックして、以下のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- プログラミングに関する簡単なヒントを集めた知識ベース
- Compiler Collection のコンポーネントのマニュアル、ソフトウェアと共にインストールされるマニュアルの訂正
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード
- 新しい技術の紹介
- `http://www.sun.co.jp/developers/` でも開発者向けのリソースが提供されています。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、以下のサイトからお問い合わせください (このマニュアルで回答されていないものに限ります)。

http://sun.co.jp/service/contacting

PART I

C++ コンパイラ

第1章

C++ コンパイラの紹介

本章では、C++ および C++ コンパイラの概要を説明しています。

標準の準拠

この C++ コンパイラ (CC) は、『ISO International Standard for C++, ISO IS 14882:1998, Programming Language - C++』に準拠しています。このリリースに含まれる README (最新情報) ファイルには、この標準の仕様と異なる記述が含まれています。

SPARC™ プラットフォームでは、このコンパイラは、UltraSPARC™ の実装と SPARC V8 と SPARC V9 の「最適化活用」機能をサポートします。これらの機能は、Prentice-Hall によって SPARC International のために出版された SPARC アーキテクチャマニュアル(トッパン刊) の バージョン 8 と SPARC Architecture Manual Version (英語版のみ) のバージョン 9 (ISBN 0-13-099227-5) に定義されています。

このマニュアルでは、「標準」は、上記の標準の各バージョンに準拠していることを意味します。「非標準」や「拡張」は、これらの標準のバージョンに準拠しない機能のことを指します。

これらの標準は、それぞれの標準を規定する組織によって改定されることがあります。したがって、コンパイラが準拠するバージョンの標準が改定されたり、まったく書き換えられた場合は、機能によっては、Sun C++ コンパイラの将来のリリースで前のリリースと互換性がなくなる場合があります。

C++ README ファイル

C++ コンパイラの README ファイルでは、コンパイラに関する重要な情報を取り上げています。これらの情報は次のとおりです。

- マニュアルの印刷後に判明した情報
- 新規および変更された機能
- ソフトウェアの非互換性
- 問題および解決方法
- 制限および互換性の問題
- 出荷可能なライブラリ
- 実装されていない規格

C++ README ファイルのテキスト版を表示するには、コマンドプロンプトで次のコマンドを入力してください。

```
example% CC -xhelp=readme
```

Netscape Communicator 4.0 (または、互換バージョン) ブラウザで HTML 版の README を表示するには、次のファイルを開きます。

```
/opt/SUNWspro/docs/ja/index.html
```

C++ コンパイラソフトウェアが `/opt` ディレクトリにインストールされていない場合、システムのどこにインストールされているのかをシステム管理者に尋ねてください。ブラウザは HTML 文書の一覧を表示します。README を開くには、一覧の上の対応するタイトルをクリックします。

マニュアルページ

オンラインのマニュアルページ (`man`) では、コマンドや関数、サブルーチン、およびその機能に関する情報を簡単に参照できます。

マニュアルページを表示するには、次のように入力してください (*topic* には、参照したいコマンドやライブラリ関数の名前を指定)。

```
example% man topic
```

C++ のマニュアルで参考情報としてマニュアルページ名を記載する場合は、名前とセクション番号が示されています。CC(1) は、man CC で表示されます。その他のセクションのマニュアルページ、たとえば ieee_flags(3M) は、man コマンドに -s オプションを使用すると表示されます。

```
example% man -s 3M ieee_flags
```

ライセンス

C++ コンパイラでは、ネットワークライセンスを使用します。これについては、『Forte Developer 7 インストールガイド』を参照してください。

ライセンスがあれば、コンパイラを起動できます。同じマシン上で同じユーザーであれば、1 ライセンスで同時に何回でもコンパイルできます。

C++ と一緒にほかのユーティリティを実行する場合には、購入したパッケージによっては、複数のライセンスが必要になる場合があります。

C++ コンパイラの新機能

C++ コンパイラで導入される新しい機能は次のとおりです。

全般的な強化機能

- テンプレートキャッシュが、不要になります。 -instances

C++ コンパイラの今回のリリースでは、テンプレートのインスタンス化機能が大幅に向上しています。デフォルトのテンプレートインスタンス化モデルを使用するモデルにおいて 1 つのディレクトリで作成できるプログラムが 1 つだけである、という制約はなくなりました。

`-instances=static` のような代替インスタンス化モデルに依存しているほとんどのプログラムは、今後、新しいデフォルトインスタンス化モデルを使用できます。

テンプレートインスタンス化機能が改良され向上したため、テンプレートキャッシュを回避すればコンパイル時間が短縮され、静的関数の重複を回避すれば実行可能ファイルのサイズが縮小されます。

詳細については、290 ページの「`-instances=a`」 を参照してください。

- 変数スコープ用にリンカーマップファイルを使用する必要がなくなりました。
  `-xldscope`

動的ライブラリのシンボルのエクスポートを 2 種類の方法で制御できるようになっています。リンカースコープと呼ばれるこの機能は、長い間、リンカーマップファイルによってサポートされてきたものです。第 1 の方法では、新しい宣言指定子をコードに埋め込めます。

`__global`、`__symbolic`、`__hidden` をコードに直接埋め込むことにより、マップファイルを使用する必要がなくなります。第 2 の方法では、`-xldscope` をコマンド行で指定することによって、変数スコープのデフォルト設定を制御できます。

360 ページの「`-xldscope={v}`」を参照してください。宣言指定子については、50 ページの「リンカースコープ」で詳しく解説しています。

- マクロのためのパワフルな新診断機能: `-xdumpmacros`

今回のリリースでは、アプリケーションで使用するマクロの動作を追跡する、2 つのプラグマと 1 つのコンパイラオプションが新たに導入されています。対象としては、システムヘッダーに定義されるマクロが含まれます。

`-xdumpmacros` オプションをコマンド行で使用すれば、マクロ定義を調べたり、プログラム内の定義されている場所、解除されている場所、使われている場所を調べることができます。を調べたりできます。焦点を絞るには、新機能である `dumpmacros` プラグマと `end_dumpmacros` プラグマをソース内で直接使用します。

344 ページの「`-xdumpmacros[=value[,value...]]`」 と 426 ページの「`#pragma dumpmacros`」.427 ページの「`#pragma end_dumpmacros`」 を参照してください。

■ VIS™ Developers Kit のサポート: `-xvis` (SPARC)

VIS 命令セット Software Developers Kit (VSDK) に定義されているアセンブリ言語テンプレートを使用しているときには、`-xvis=[yes|no]` オプションを使用します。デフォルトは、`-xvis=no` です。

VIS 命令セットは、SPARC v9 命令セットの拡張機能です。UltraSPARC プロセッサは 64 ビットですが、データセットサイズが 8 ビットや 16 ビットに制限されていることが少なくありません。マルチメディアアプリケーションではとくにその傾向が強くなります。VIS 命令は 1 つの命令で 16 ビットデータを 4 個処理できるため、イメージング、線形代数、シグナル処理、オーディオ、ビデオ、ネットワーキングといった新しいメディアを扱うアプリケーションのパフォーマンスが大きく向上します。

VSDK の詳細については、http://www.sun.com/processors/vis を参照してください。418 ページの「`-xvis[={yes|no}]`」を参照してください。

■ C99 実行時ライブラリと環境のサポート `-xlang=c99`

C99 標準 (ISO/IEC 9899:1999, Programming Language - C) をサポートするオペレーティングシステムの場合、`-xlang=c99` は、C ライブラリ関数を呼び出す C と C++ のコードの C99 実行時動作を指定します。C complex 型のような一部の C99 動作は、`-xc99=%all` オプションを C コンパイラで使用することに依存しており、printf をはじめとする他の C 動作はこの点に依存していません。

注 – C99 サポートは、compat モード (`-compat=4`) では利用できません。

---

358 ページの「`-xlang=language[,language]`」を参照してください。

■ UTF-16 文字列リテラルのサポート `-xustr`

ISO10646 UTF-16 文字列リテラルを使用する国際化アプリケーションをサポートする必要がある場合には、`-xustr=ascii_utf16_ushort` を指定します。つまり、16 ビット文字で構成される文字列リテラルがコードに含まれている場合には、このオプションを使用します。このオプションが指定されていない場合、コンパイラは 16 ビット文字列リテラルの生成、認識のいずれも行いません。このオプションを使用することによって、U"*ASCII_string*" 文字列リテラルは unsigned short 型の配列として認識されます。このような文字列はまだ規格の一部となっていないので、このオプションは、非標準 C++ の認識を可能にします。

417 ページの「`-xustr={ascii_utf16_ushort|no}`」を参照してください。

■ OpenMP の拡張サポート `-xopenmp`

明示的な並列化用の OpenMP インタフェースは、従来どおりに C++ コンパイラに実装されています。

以下を実現する OpenMP 機能が、C++ コンパイラに新たに追加されました。

- クラスオブジェクトを OpenMP データ節で使用できる。
- OpenMP プラグマをクラスメンバー関数で使用できる。

OpenMP のコンパイラコマンドの詳細は、379 ページの「`-xopenmp[=i]`」 を参照してください。多重処理アプリケーションを構築するための OpenMP C++ アプリケーションプログラムインタフェース (API) の詳細は、『OpenMP API ユーザーズガイド』を参照してください。

■ 改良版 `-xprofile` (SPARC)

`-xprofile` オプションは、次のように改良されました。

- 共有ライブラリのプロファイル処理のサポート
- -xprofile=collect -mt による、スレッドで使用しても安全なプロファイル収集
- 複数のプログラムを、単一のプロファイルディレクトリにプロファイル処理するサポートの改良

`-xprofile=use` により、一意でないベース名を持つ複数のオブジェクトファイルのデータが入っている複数のプロファイルディレクトリの中で、コンパイラがプロファイルデータを見つけることができます。オブジェクトファイルのプロファイルデータをコンパイラが見つけられない場合には、コンパイラの新しいオプション `-xprofile_pathmap=`*collect-prefix* : *use-prefix* を使用します。

397 ページの「`-xprofile=p`」 と 401 ページの「`-xprofile_pathmap`」 を参照してください。

コンパイルの高速化

■ 構文チェックの高速化: `-xe`

`-xe` を指定すると、コンパイラは構文と意味のエラーだけをチェックし、オブジェクトコードを生成しません。

コンパイルによってオブジェクトファイルを生成する必要がない場合には、-xe オプションを使用してください。たとえば、コードの一部を削除することによってエラーメッセージの原因を切り分ける場合には、-xe を使用することによって編集とコンパイルを高速化できます。

349 ページの「-xe」を参照してください。

- -xprofile_ircache による高速プロファイル処理

-xprofile_ircache[=*path*] と -xprofile=collect|use を併用すれば、収集フェーズで保存したコンパイルデータを再使用することによって、use フェーズでのコンパイル時間を短縮できます。

大きなプログラムの場合、中間データが保存されているので、use 段階でのコンパイル時間を大幅に短縮できます。保存データによって、ディスクスペース要求がある程度増えることに注意してください。

400 ページの「-xprofile_ircache[=path]」を参照してください。

- 冗長なテンプレートインスタンス生成の停止 -instlib=*filename*

-instlib=*filename* を使用すると、ライブラリと現在のオブジェクトとで重複するテンプレートインスタンスの生成が禁止されます。一般に、大量のインスタンスをライブラリと共有しているプログラムの場合には -instlib=*filename* を試し、コンパイル時間が短縮されるかどうかを確認してください。

291 ページの「-instlib=<出力ファイル>」を参照してください。

- -template=geninlinefuncs による関数の生成

通常、関数が呼び出され、かつインライン展開できない場合を除き、C++ コンパイラはインラインテンプレート関数を生成しません。ただし、-template=geninlinefuncs を指定すれば、以前は生成されていなかった、明示的にインスタンス化されたクラステンプレートのインラインメンバー関数を、コンパイラがインスタンス化します。これらの関数のリンケージは、いかなる場合もローカルです。

315 ページの「-template=opt[,opt...]」を参照してください。

- プリコンパイル済みヘッダー -xpch

コンパイラの今回のリリースでは、プリコンパイル済みヘッダー機能が新たに導入されています。プリコンパイル済みヘッダーファイルは、大量のソースコードが入っている一連の共通インクルードファイル群を共有するソースファイルを持つアプリケーションの、コンパイル時間を短縮することを目的としています。プリコンパイル済みヘッダーは、1 つのソースファイルに入っている一連のヘッダーファイルに関する情

報を収集し、このソースファイルを再コンパイルするとき、および、同じ一連のヘッダーを持つ他のソースファイルをコンパイルするときに、収集された情報を使用するという仕組みになっています。この機能を利用するには、-xpch と -xpchstop のオプションを #pragma hdrstop 指令とともに使用します。

383 ページの「-xpch=v」、387 ページの「-xpchstop=file」 と 428 ページの「#pragma hdrstop」 を参照してください。

- -xjobs=*n* (SPARC) による複数のプロセッサの使用

コンパイラが処理を行うために生成するプロセスの数を設定するには、-xjobs=*n* オプションを指定します。このオプションにより、複数の CPU が搭載されているマシンでの構築時間を短縮できます。現在、-xjobs が動作するのは、-xipo オプションを指定した場合だけです。-xjobs=*n* を指定すると、内部手続きオプティマイザは、複数のファイルをコンパイルするために呼び出すコードジェネレータの最大数として、*n* を使用します。

358 ページの「-xjobs=n」を参照してください。

移植の簡略化

- 移植の簡略化: `-xmemalign`

データの境界整列についてコンパイラが使用する想定を制御するには、`-xmemalign` オプションを使用します。境界整列が潜在的に正しくないメモリーアクセスにつながる生成コードを制御し、境界整列が正しくないアクセスが発生したときのプログラム動作を制御すれば、より簡単に Solaris オペレーティング環境にコードを移植できます。

369 ページの「`-xmemalign=ab`」を参照してください。

- char の符号の設定: `-xchar`

`-xchar[={signed|s|unsigned|u}]` は、char 型が符号なしと定義されているシステムからコードを簡単に移行できるようにするためのものです。こういったシステムから移行するのでないかぎり、このオプションは使用しないでください。符号付きまたは符号なしであると明示的に書き直す必要があるのは、符号に依存するコードだけです。

336 ページの「`-xchar[=o]`」を参照してください。

- 移植したコードのデバッグ: `-xport64`

64 ビット環境へのコード移植では、新機能である `-xport64` オプションを使用してください。このオプションは、具体的には V7 (ILP32) などの 32 ビットアーキテクチャを V9 (LP64) などの 64 ビットアーキテクチャにコード移植する際によく見られる、値 (ポインタを含む) の切り捨て、符号拡張、ビット配置の変更といった問題について警告します。

389 ページの「`-xport64[=(v)]`」を参照してください。

パフォーマンスの向上

- リンカーサポートされたスレッドローカルデータ記憶装置による実行時パフォーマンスの向上: `-xthreadvar` (SPARC)

コンパイラに新たに追加された、リンカーサポートによるスレッドローカル記憶装置を使用すれば、以下を行えます。

- POSIX インタフェースの高速実装を活用してスレッド固有データを割り当てる。
- マルチプロセスプログラムをマルチスレッドプログラムに変換する。

- スレッドローカルな記憶装置を使用する Windows アプリケーションを Solaris オペレーティング環境に移植する。
- OpenMP のスレッド非公開変数の高速実装を活用する。

スレッドローカル変数を宣言することによって、スレッドローカルな記憶装置をコンパイラで利用することができるようになりました。通常の変数宣言に変数指定子 `__thread` を追加したものと、コマンド行オプション `-xthreadvar` の 2 つを併記します。

414 ページの「`-xthreadvar[=o]`」を参照してください。宣言指定子については、51 ページの「スレッドローカルな記憶装置」で詳しく解説しています。

- ページフォルト削減による実行時パフォーマンスの向上: `-xF`

新機能 `-xF` を使用すれば、リンカーによる変数と関数の最適な順序への並べ替えが可能になります。これにより、実行時パフォーマンスを低下させる次の問題を解決できます。

- 関連性のない複数の変数がメモリー内で近接することによって引き起こされる、キャッシュとページの競合
- 関連性のある複数の変数が、メモリー内で近接していないことによってもたらされる、不必要に大きい作業セットのサイズ
- 効果的なデータ密度を低下させる弱い変数の使用されないコピーによってもたらされる、不必要に大きい作業セットのサイズ

349 ページの「`-xF[=v[,v...]]`」を参照してください。

- 新しいプラグマによる実行時パフォーマンスの向上

C++ コンパイラには、コードの最適化を向上させる 4 種類の新しいプラグマが備わっています。

- 424 ページの「`#pragma does_not_read_global_data`」
- 425 ページの「`#pragma does_not_return`」
- 425 ページの「`#pragma does_not_write_global_data`」
- 432 ページの「`#pragma rarely_called`」

421 ページの「プラグマ」を参照してください。

- リンクオプティマイザによる実行時パフォーマンスの向上: `-xlinkopt`

C++ コンパイラで `-xlinkopt` コマンドを指定することによって、再配置可能なオブジェクトファイルに対して、リンク時の最適化が行われるようになりました。

-xlinkopt を指定することにより、コンパイラはリンク時に追加の最適化を行います。リンクされる .o ファイルは変更されません。最適化は、実行可能プログラムに対してのみ行われます。-xlinkopt オプションがその効果をもっとも発揮するのは、プログラム全体をプロファイルフィードバックを使ってコンパイルするときです。

365 ページの「-xlinkopt[=level]」を参照してください。

- -xpagesize=*n* (SPARC) による実行時パフォーマンスの向上

-xpagesize=*n* オプションは、スタックおよびヒープの優先ページサイズの設定に使用します。*n* に指定できる値は、8K、64K、512K、4M、32M、256M、2G、16G、または default です。対象となるプラットフォーム上の Solaris オペレーティング環境で有効なページサイズを指定する必要があります。getpagesize(3C) を使用するとそのサイズがわかります。有効なページサイズを指定しなかった場合、その要求は実行時に無視され、メッセージ出力は行われません。pmap(1) または meminfo(2) を使用すれば、ターゲットプラットフォームのページサイズを確認できます。

注 – この機能を利用できるのは、Solaris 9 オペレーティング環境においてだけです。このオプションを使用してコンパイルしたプログラムは、旧バージョンの Solaris オペレーティング環境ではリンクされません。

このオプションは、-xpagesize_stack と -xpagesize_heap のマクロです。

380 ページの「-xpagesize=n」、381 ページの「-xpagesize_heap=n」 と 382 ページの「-xpagesize_stack=n」 を参照してください。

警告とエラーの新しい制御機能

- -erroff による警告メッセージのフィルタリング

新しいオプション -erroff は、コンパイラのフロントエンドからの警告メッセージを抑止します。エラーメッセージとドライバからのメッセージに対しては作用しません。-erroff を使って、特定の警告メッセージだけを抑止したり出力したりすることも可能です。

たとえば、-erroff=*tag* は、この *tag* が示す警告メッセージを抑止します。一方、-erroff=%all,no%*tag* は、*tag* が示すメッセージ以外の警告メッセージをすべて抑止します。警告メッセージのタグを表示するには、-errtags=yes オプションを使用します。

259 ページの「-erroff[=t]」を参照してください。

■ -errtags と -errwarn によるコンパイルの中断

-errtags と -errwarn のコンパイラオプションは、コンパイラが特定の警告を出力したときにコンパイルを停止します。特定の警告のタグを見つけるため -errtags=yes を設定し、-errwarn=*tag* を指定します。*tag* は、-errtags が返してきた、目的の警告メッセージの一意の識別子です。

-errwarn=%all を指定することによって、すべての警告出力時にコンパイルを停止させることができます。

260 ページの「-errtags[=a]」 と 261 ページの「-errwarn[=t]」 を参照してください。

■ -filt=[no%]stdlib による標準ライブラリ名のフィルタリングの向上

デフォルト時に設定される -filt=[no%]stdlib オプションは、リンカーとコンパイラの両方のエラーメッセージに出力される標準ライブラリからの名前を簡略化します。この結果、標準ライブラリ関数の名前を容易に認識できるようになります。このフィルタリングを無効にするには、-filt=no%stdlib を指定します。

270 ページの「-filt[=filter[,filter...]]」を参照してください。

C++ ユーティリティ

現在、ほとんどの C++ ユーティリティは従来の UNIX® ツールに統合され、オペレーティングシステムに含まれています。

- lex: テキストの単純な字句解析に使用するプログラムを生成する。
- yacc: 構文に応じて入力ストリームを解析するための C 関数を生成する。
- prof: プログラム内のモジュールの実行プロファイルを作成する。
- gprof: プログラムの実行時パフォーマンスについての手続き単位のプロファイル。
- tcov: プログラムの実行時パフォーマンスについての文単位のプロファイル。

これら UNIX ツールについての詳細は、『プログラムのパフォーマンス解析』や関連するマニュアルページを参照してください。

各国語のサポート

本バージョンの C++ では、英語以外の言語を使用したアプリケーションの開発をサポートしています。対象としている言語は、ヨーロッパのほとんどの言語と日本語です。このため、アプリケーションをある言語から別の言語に簡単に置き換えることができます。この機能を国際化と呼びます。

通常 C++ コンパイラでは、次のように国際化を行なっています。

- どの国のキーボードから入力された ASCII 文字でも認識する (つまりキーボードに依存せず、8 ビット透過となっています)
- メッセージによっては現地語で出力できるものもある
- 注釈、文字列、データに、現地語の文字を使用できる
- C++ は、Extended UNIX Character (EUC) 準拠の文字セットをサポートしています。この文字セットでは、文字列中のすべての NULL バイトが NULL 文字になります。また、文字列中で ASCII 値が '/' のバイトはすべて '/' 文字になります。

変数名は国際化できません。必ず英語の文字を使用してください。

アプリケーションをある国の言語から別の国の言語に変更するには、ロケールを設定します。言語の切り換えのサポートに関する情報については、オペレーティング環境のマニュアルを参照してください。

第2章

C++ コンパイラの使用方法

この章では、C++ コンパイラの使用方法を説明します。

コンパイラの主な目的は、C++ などの高水準言語で書かれたプログラムをコンピュータハードウェアで実行できるデータファイルに変換することです。C++ コンパイラでは次のことができます。

- ソースファイルを再配置可能なバイナリ (-.o) ファイルに変換する。 これらのファイルはその後、実行可能ファイル、(-xar オプションで) 静的 (アーカイブ) ライブラリ (.a) ファイル、動的 (共有) ライブラリ (.so) ファイルなどにリンクされます。
- オブジェクトファイルとライブラリファイルのどちらか (または両方) をリンク (または再リンク) して実行可能ファイルを作成する。
- 実行時デバッグを (-g オプションで) 有効にして、実行可能ファイルをコンパイルする。
- 文レベルや手続きレベルの実行時プロファイルを (-pg オプションで) 有効にして、実行可能ファイルをコンパイルする。

コンパイル方法の概要

この節では、C++ コンパイラを使って C++ プログラムのコンパイルと実行をどのように行うかを簡単に説明します。コマンド行オプションの詳しい説明については、付録 A を参照してください。

注 – この章のコマンド行の例は、CC の使用方法を示すためのものです。実際に出力される内容はこれと多少異なる場合があります。

C++ アプリケーションを構築して実行するには、基本的に次の手順が必要です。

1. エディタで C++ソースファイルを作成する。このソースファイルには、表 2-1 に列挙されている接尾辞のいずれかを使用します。
2. コンパイラを起動して実行可能ファイルを作成する。
3. 実行可能ファイルの名前を入力してプログラムを実行する。

次のプログラムは、メッセージを画面に表示する例です。

```
example% cat greetings.cc
    #include <iostream>
    int main()  {
      std::cout << "Real programmers write C++!" << std::endl;
      return 0;
    }
example% CC greetings.cc
example% a.out
 Real programmers write C++!
example%
```

この例では、ソースファイル `greetings.cc` を CC でコンパイルしています。デフォルトでは、実行可能ファイルがファイル `a.out` として作成されます。プログラムを起動するには、コマンドプロンプトで実行可能ファイル名 `a.out` を入力します。

従来、UNIX コンパイラは実行可能ファイルに `a.out` という名前を付けていました。しかし、すべてのコンパイルで同じファイルを使用するのは不都合な場合があります。そのファイルがすでにあれば、コンパイラを実行したときに上書きされてしまうからです。次の例のように、コンパイラオプションに `-o` を使用すれば、実行可能出力ファイルの名前を指定できます。

```
example% CC -o greetings greetings.C
```

この例では、-o オプションを指定することによって、実行可能なコードがファイル greetings に書き込まれます (プログラムにソースファイルが 1 つだけしかない場合は、ソースファイル名から接尾辞を除いたものを出力ファイル名にすることが一般的です)。

あるいは、コンパイルの後に mv コマンドを使って、デフォルトの a.out ファイルを別の名前に変更することもできます。いずれの場合も、プログラムを実行するには、実行可能ファイルの名前を入力します。

```
example% greetings
Real programmers write C++!
example%
```

コンパイラの起動

この後の節では、CC コマンドで使用する規約、コンパイラのソース行指令など、コンパイラの使用に関連する内容について説明します。

コマンド構文

コンパイラの一般的なコマンド行の構文を次に示します。

```
CC [options] [source-files] [object-files] [libraries]
```

options は、先頭にダッシュ (-) またはプラス記号 (+) の付いたキーワード (オプション) です。このオプションには、引数をとるものがあります。

通常、コンパイラオプションの処理は、左から右へと行われ、マクロオプション (他のオプションを含むオプション) は、条件に応じて内容が変更されます。ほとんどの場合、同じオプションを 2 回以上指定すると、最後に指定したものだけが有効になり、オプションの累積は行われません。次の点に注意してください。

- すべてのリンカーオプション、ならびに -features、-I、 -l、-L、-library、-pti、-R、-staticlib、-U、-verbose および -xprefetch オプションで指定した内容は蓄積され、上書きはされません。
- -U オプションは、すべての -D オプションの後に処理される。

ソースファイル、オブジェクトファイル、およびライブラリは、コマンド行に指定した順にコンパイルとリンクが行われます。

次の例では、CC を使って 2 つのソースファイル (growth.C と fft.C) をコンパイルし、実行時デバッグを有効にして growth という名前の実行可能ファイルを作成します。

```
example% CC -g -o growth growth.C fft.C
```

ファイル名に関する規則

コンパイラがコマンド行に指定されたファイルをどのように処理するかは、ファイル名に付加された接尾辞で決まります。次の表以外の接尾辞を持つファイルや、接尾辞がないファイルはリンカーに渡されます。

表 2-1 C++ コンパイラが認識できるファイル名接尾辞

接尾辞	言語	処理
.c	C++	C++ ソースファイルとしてコンパイルし、オブジェクトファイルを現在のディレクトリに入れる。オブジェクトファイルのデフォルト名は、ソースファイル名に .o 接尾辞が付いたものになる。
.C	C++	.c 接尾辞と同じ処理。
.cc	C++	.c 接尾辞と同じ処理。
.cpp	C++	.c 接尾辞と同じ処理。
.cxx	C++	.c 接尾辞と同じ処理。
.c++	C++	.c 接尾辞と同じ処理。
.i	C++	C++ ソースファイルとして扱われるプリプロセッサ出力ファイル。 .c 接尾辞と同じ処理。
.s	アセンブラ	ソースファイルをアセンブラを使ってアセンブルする。
.S	アセンブラ	C 言語プリプロセッサとアセンブラを使ってソースファイルをアセンブルする。

表 2-1　C++ コンパイラが認識できるファイル名接尾辞 (続き)

接尾辞	言語	処理
.il	インライン展開	アセンブリ用のインラインテンプレートファイルを使ってインライン展開を行う。コンパイラはテンプレートを使って、選択されたルーチンのインライン呼び出しを展開する (インラインテンプレートファイルは、特殊なアセンブラファイルです。inline(1) のマニュアルページを参照してください)。
.o	オブジェクトファイル	オブジェクトファイルをリンカーに渡す。
.a	静的 (アーカイブ) ライブラリ	オブジェクトライブラリの名前をリンカーに渡す。
.so .so.n	動的 (共有) ライブラリ	共有オブジェクトの名前をリンカーに渡す。

複数のソースファイルの使用

C++ コンパイラでは、複数のソースファイルをコマンド行に指定できます。コンパイラが直接または間接的にサポートするファイルも含めて、コンパイラによってコンパイルされる 1 つのソースファイルを「コンパイル単位」といいます。C++ では、それぞれのソースが別個のコンパイル単位として扱われます。

バージョンが異なるコンパイラでのコンパイル

C++ 5.1 以降のコンパイラは、テンプレートキャッシュディレクトリにテンプレートキャッシュのバージョンを示す文字列を付けます。

このコンパイラは、デフォルト時にはキャッシュを使用しません。キャッシュを使用するのは、-instances=extern が指定されているときだけです。キャッシュを使用する場合、コンパイラはキャッシュディレクトリのバージョンを調べ、その結果キャッシュバージョンに問題があることがわかると、エラーメッセージを出力します。将来の C++ コンパイラもキャッシュのバージョンを調べます。たとえば、将来の

コンパイラは異なるテンプレートキャッシュのバージョン識別子を持っているため、現在のリリースで作成されたキャッシュディレクトリを処理しようとすると、次のようなエラーを出力します。

```
SunWS_cache:エラー:/SunWS_cache のテンプレートデータベースは、
このコンパイラと互換性がありません。
```

同様に、現在のリリースのコンパイラで以降のバージョンのコンパイラで作成されたキャッシュディレクトリを処理しようとすると、エラーが発行されます。

C++ コンパイラ 5.0 で作成されたテンプレートキャッシュディレクトリにはバージョン識別子が付けられていません。しかし、現在のリリースのコンパイラは、5.0 のキャッシュディレクトリをエラーや警告なしに処理できます。これは、C++ コンパイラが 5.0 のキャッシュディレクトリを、使用できるディレクトリ形式に変換するためです。

C++ 5.0 コンパイラは、5.0 より新しいリリースのコンパイラが作成したキャッシュディレクトリを使用できません。C++ コンパイラ 5.0 は形式の違いを認識できず、C++ コンパイラ 5.1 (または、これ以降のリリース) で作成されたキャッシュディレクトリを処理しようとすると、エラーを発行します。

コンパイラをアップグレードする際には、必ずキャッシュを消去するようにするとよいでしょう。テンプレートキャッシュディレクトリ (ほとんどの場合、テンプレートキャッシュディレクトリの名前は`SunWS_cache` です) が入っているディレクトリすべてに対し、`CCadmin -clean` を実行します。`CCadmin -clean`の代わりに、`rm -rf SunWS_cache` と指定しても同様の結果が得られます。テンプレートのクリアー方法の最新情報については、`http://forte.sun.com/s1scc/articles/index.html` にある 『Upgrading Your C++ Compiler』を参照してください。

コンパイルとリンク

この節では、プログラムのコンパイルとリンクについていくつかの側面から説明します。次の例では、CC を使って 3 つのソースファイルをコンパイルし、オブジェクトファイルをリンクして prgrm という実行可能ファイルを作成します。

```
example% CC file1.cc file2.cc file3.cc -o prgrm
```

コンパイルとリンクの流れ

前の例では、コンパイラがオブジェクトファイル (file1.o、file2.o、file3.o) を自動的に生成し、次にシステムリンカーを起動してファイル prgrm の実行可能プログラムを作成します。

コンパイル後も、オブジェクトファイル (file1.o、file2.o、file3.o) はそのまま残ります。この規則のおかげで、ファイルの再リンクと再コンパイルを簡単に行えます。

注 – ソースファイルが 1 つだけであるプログラムに対してコンパイルとリンクを同時に行なった場合は、対応する .o ファイルが自動的に削除されます。複数のソースファイルをコンパイルする場合を除いて、すべての .o ファイルを残すためにはコンパイルとリンクを別々に行なってください。

コンパイルが失敗すると、エラーごとにメッセージが返されます。エラーがあったソースファイルの .o ファイルは生成されず、実行可能プログラムも作成されません。

コンパイルとリンクの分離

コンパイルとリンクは別々に行うことができます。-c オプションを指定すると、ソースファイルがコンパイルされて .o オブジェクトファイルが生成されますが、実行可能ファイルは作成されません。-c オプションを指定しないと、コンパイラはリンカー

を起動します。コンパイルとリンクを分離すれば、1 つのファイルを修正するためにすべてのファイルを再コンパイルする必要はありません。次の例では、最初の手順で 1 つのファイルをコンパイルし、次の手順でそれを他のファイルとリンクします。

```
example% CC -c file1.cc                          <- 新しいオブジェクトファイルを作成する。
example% CC -o prgrm file1.o file2.o file3.o  <- 実行可能ファイルを作成する。
```

リンク時には、完全なプログラムを作成するのに必要なすべてのオブジェクトファイルを指定してください。オブジェクトファイルが足りないと、リンクは「undefined external reference (未定義の外部参照がある)」エラー (ルーチンがない) で失敗します。

コンパイルとリンクの整合性

コンパイルとリンクを別々に実行する場合で、次のコンパイラオプションを使用する場合は、コンパイルとリンクの整合性を保つことが非常に重要です。

- `B`
- `-compat`
- `-fast`
- `-g`
- `-g0`
- `-library`
- `misalign`
- `-mt`
- `-p`
- `-xa`
- `-xarch`
- `-xcg92` および `-xcg89`
- `xipo`
- `-xpagesize`
- `-xpg`
- `-xprofile`
- `-xtarget`

これらのオプションのいずれかを使用してサブプログラムをコンパイルした場合は、リンクでも同じオプションを使用してください。

- -library、-fast、-xtarget、-xarch オプションの場合、コンパイルとリンクを同時に行えば渡されるはずのリンカーオプションも含める必要があります。
- -p、-xpg、-xprofile オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プロファイル処理ができなくなります。
- -g、-g0 オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プログラムを正しくデバッグできなくなります。これらのオプションでコンパイルされず、-g または -g0 でリンクされるいずれのモジュールも、デバッグには使用できません。-g または -g0オプション付きの main 関数を含むモジュールをコンパイルすることは、通常はデバッグに必要になります。

次の例では、-xcg92 コンパイラオプションを使用してプログラムをコンパイルしています。このオプションは -xtarget=ss1000 用のマクロであり、-xarch=v8 -xchip=super -xcache=16/64/4:1024/64/1 と展開されます。

```
example% CC -c -xcg92 sbr.cc
example% CC -c -xcg92 smain.cc
example% CC -xcg92 sbr.o smain.o
```

プログラムがテンプレートを使用する場合は、リンク時にその中のいくつかがインスタンス化される可能性があります。その場合、インスタンス化されたテンプレートは最終行 (リンク行) のコマンド行オプションを使用してコンパイルされます。

SPARC V9 のためのコンパイル

64 ビットオブジェクトのコンパイル、リンク、実行には、V9 SPARC の Solaris 7 または Solaris 8 のオペレーティング環境で 64 ビットカーネルが動作していなければなりません。64 ビットのコンパイルは、-xarch=v9 オプション、-xarch=v9a オプション、-xarch=v9b オプションで指定します。

コンパイラの診断

-verbose オプションを使用すると、呼び出される名前やバージョン番号および各コンパイル段階のコマンド行など、プログラムのコンパイル中に役立つ情報を表示できます。

コマンド行に指定された引数をコンパイラが認識できない場合には、それらはリンカーオプション、オブジェクトプログラムファイル名、ライブラリ名のいずれかとみなされます。

基本的な区別は次のとおりです。

- 認識できないオプション (先頭にダッシュ (–) かプラス符号 (+) の付いたもの) には、警告が生成されます。
- 認識できない非オプション (先頭にダッシュかプラス符号 (+) が付いていないもの) には、警告が生成されません(ただし、リンカーへの引き渡しは行われます。リンカーも認識できないと、リンカーからエラーメッセージが生成されます)。

次の例で、-bit は CC によって認識されないため、リンカー (ld) に渡されます。リンカーはこれを解釈しようとします。単一文字の ld オプションは連続して指定できるので、リンカーは -bit を -b、-i、-t とみなします。これらはすべて有効な ld オプションです。しかし、これは本来の意図とは異なります。

```
example% CC -bit move.cc              <- -bit は CC オプションとして認識
されない

CC: 警告: ld が起動される場合は、オプション-bit は ld に渡されます。それ
以外は無視されます。
```

次の例では、CC オプション -fast を指定しようとしましたが、先頭のダッシュ (-) を入力しませんでした。コンパイラはこの引数もリンカーに渡します。リンカーはこれをファイル名とみなします。

```
example% CC fast move.cc              <- ユーザーは -fast と入力したつも
りだった
move.CC:
ld: 重大なエラー: ファイル fast: ファイルもディレクトリもありません。
ld: 重大なエラー: ファイル処理エラー。a.out へ書き込まれる出力がありません。
```

コンパイラの構成

C++ コンパイラパッケージは、フロントエンド (CC コマンド本体)、オプティマイザ (最適化)、コードジェネレータ (コード生成)、アセンブラ、テンプレートのプリリンカー (リンクの前処理をするプログラム)、リンクエディタから構成されています。コマンド行オプションで他の指定を行わないかぎり、CC コマンドはこれらの構成要素をそれぞれ起動します。

これらの構成要素はいずれもエラーを生成する可能性があり、構成要素はそれぞれ異なる処理を行うため、エラーを生成した構成要素を識別することがエラーの解決に役立つことがあります。それには、-v オプションと -dryrun オプションを使用します。

次の表に示すように、コンパイラの構成要素への入力ファイルには異なるファイル名接尾辞が付いています。どのようなコンパイルを行うかは、この接尾辞で決まります。ファイル名接尾辞の意味については、表 2-1を参照してください。

表 2-2 C++ コンパイルシステムの構成要素

構成要素	内容の説明	使用時の注意
ccfe	フロントエンド (コンパイラプリプロセッサ (前処理系) とコンパイラ)	
iropt	SPARC:コードオプティマイザ	-xO[2-5], -fast
ir2hf	IA:中間言語トランスレータ	-xO[2-5], -fast
inline	SPARC:アセンブリ言語テンプレートのインライン展開	.il ファイルを指定
ube_ipa	IA:内部手続きアナライザ	-xO4、-xO5、あるいは -fast 付きの -xcrossfile=1
fbe	アセンブラ	
cg	SPARC:コード生成、インライン機能、アセンブラ	
ube	IA:コードジェネレータ	-xO[2-5], -fast
CClink	テンプレートのプリリンカー	
ld	従来のリンクエディタ	
ild	インクリメンタルリンクエディタ	-g, -xildon

注 – このマニュアルで "IA" とは Intel 32 ビットプロセッサアーキテクチャーのことです。このアーキテクチャーには、Pentium、Pentium Pro、および Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサおよび AMD 社と Cyrix 社製の互換マイクロプロセッサチップがあります。

指示および名前の前処理

この節では、C++ コンパイラ特有の前処理の指示について説明します。

プラグマ

プリプロセッサキーワードの `pragma` は、C++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。C++ コンパイラで認識されるプラグマのリストについては、付録 B を参照してください。

可変数の引数をとるマクロ

C++ コンパイラでは次の書式の`#define` プリプロセッサの指示を受け入れます。

```
#define identifier (...) replacement_list
#define identifier (identifier_list, ...) replacement_list
```

マクロパラメータリストの終わりが省略符号である場合、マクロパラメータより多くの引数をマクロの呼び出しで使用できます。追加の引数は、マクロ交換リストにおいて `__VA_ARGS__` という名前で参照できる、コンマを含んだ単一文字列にまとめられます。次の例は、変更可能な引数リストマクロの使い方を示しています。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

この結果は、次のとおりとなります。

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

事前に定義されている名前

付録の表 A-3は、事前に定義されているマクロを示しています。これらの値は、`#ifdef` のようなプリプロセッサに対する条件式の中で使用できます。`+p` オプションを指定すると、`sun`、`unix`、`sparc`、および `i386` の事前定義マクロは自動的に定義されません。

#error

`#error` 指令は、警告生成後にコンパイルを続行しなくなりました。以前の #error 指令は、警告を生成してコンパイルを続行していました。新しい #error では、他のコンパイラとの整合性が確保され、エラーメッセージを生成してコンパイルをすぐに停止するようになりました。コンパイラは終了して障害をレポートします。

メモリー条件

コンパイルに必要なメモリー量は、次の要素によって異なります。

- 各手続きのサイズ
- 最適化のレベル
- 仮想メモリーに対して設定された限度
- ディスク上のスワップファイルのサイズ

SPARC プラットフォームでメモリーが足りなくなると、オプティマイザは最適化レベルを下げて現在の手続きを実行することでメモリー不足を補おうとします。それ以後のルーチンについては、コマンド行の -xOlevel オプションで指定した元のレベルに戻ります。

1 つのファイルに多数のルーチンが入っている場合、それをコンパイルすると、メモリーやスワップ領域が足りなくなることがあります。コンパイラがメモリー不足状態になった場合には、最適化レベルを下げてみてください。あるいは、複数のルーチンソースファイルを 1 ファイルに 1 ルーチンずつ入れて分割します。

スワップ領域のサイズ

現在のスワップ領域は `swap -s` コマンドで表示できます。詳細は、`swap`(1M) のマニュアルページを参照してください。

`swap` コマンドを使った例を次に示します。

```
example% swap -s
total: 40236k bytes allocated + 7280k reserved = 47516k used,
1058708k available
```

スワップ領域の増加

ワークステーションのスワップ領域を増やすには、`mkfile`(1M) と `swap`(1M) コマンドを使用します (そのためには、スーパーユーザーでなければなりません)。`mkfile` コマンドは特定サイズのファイルを作成し、`swap -a` はこのファイルをシステムのスワップ領域に追加します。

```
example# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
example# /usr/sbin/swap -a  /home/swapfile
```

仮想メモリーの制御

1 つの手続きが数千行からなるような非常に大きなルーチンを `-xO3` 以上でコンパイルすると、大容量のメモリーが必要になることがあります。このようなときには、システムのパフォーマンスが低下します。これを制御するには、1 つのプロセスで使用できる仮想メモリーの量を制限します。

`sh` シェルで仮想メモリーを制限するには、`ulimit` コマンドを使用します。詳細は、`sh`(1) のマニュアルページを参照してください。

次の例では、仮想メモリーを 16M バイトに制限しています。

```
example$ ulimit -d 16000
```

csh シェルで仮想メモリーを制限するには、limit コマンドを使用します。詳細は、csh(1) のマニュアルページを参照してください。

次の例でも、仮想メモリーを 16M バイトに制限しています。

```
example% limit datasize 16M
```

どちらの例でも、オプティマイザは データ空間が 16M バイトになった時点でメモリー不足が発生しないような手段をとります。

仮想メモリーの限度は、システムの合計スワップ領域の範囲内でなければなりません。さらに実際は、大きなコンパイルが行われているときにシステムが正常に動作できるだけの小さい値でなければなりません。

スワップ領域の半分以上がコンパイルによって使用されることがないようにしてください。

スワップ領域が 32M バイトなら次のコマンドを使用します。

sh シェルの場合

```
example$ ulimit -d 16000
```

csh シェルの場合

```
example% limit datasize 16M
```

最適な設定は、必要な最適化レベルと使用可能な実メモリーと仮想メモリーの量によって異なります。

メモリー条件

ワークステーションには、少なくとも 64M バイトのメモリーが必要です。推奨は 128M バイトです。

実際のメモリーを調べるには、次のコマンドを使用します。

```
example% /usr/sbin/dmesg | grep mem
mem = 655360K (0x28000000)
avail mem = 602476544
```

コマンドの簡略化

CCFLAGS 環境変数で特別なシェル別名を定義するかmakeを使用すれば、複雑なコンパイラコマンドを簡略化できます。

C シェルでの別名の使用

次の例では、頻繁に使用するオプションをコマンドの別名として定義します。

```
example% alias CCfx "CC -fast -xnolibmil"
```

次に、この別名 CCfx を使用します。

```
example% CCfx any.C
```

上記のコマンド CCfx は、次のコマンドを実行するのと同じことです。

```
example% CC -fast -xnolibmil any.C
```

CCFLAGS によるコンパイルオプションの指定

CCFLAGS 環境変数を設定すると、一度に特定のオプションを指定できます。

CCFLAGS 変数は、コマンド行に明示的に指定できます。次の例は、CCFLAGS の設定方法を示したものです (C シェル)。

```
 example% setenv CCFLAGS '-xO2 -xsb'
```

次の例では、CCFLAGS を明示的に使用しています。

```
example% CC $CCFLAGS any.cc
```

make を使用する場合、CCFLAGS 変数が上の例のように設定され、makefile のコンパイル規則が暗黙的に使用された状態で make を呼び出すと、次のコンパイルが行われます (files は、複数のファイル名を示します)。

```
CC -xO2 -xsb files...
```

make の使用

make ユーティリティは、サンのすべてのコンパイラで簡単に使用できる非常に強力なプログラム開発ツールです。詳細については make(1S) のマニュアルページを参照してください。

make での CCFLAGS の使用

makefile の暗黙のコンパイラ規則を使用する (つまり、C++ コンパイル行がない) 場合は、make プログラムによって CCFLAGS が自動的に使用されます。

makefile への接尾辞の追加

makefile に別のファイルの接尾辞を追加すると、C++ にその接尾辞を取り込むことができます。次の例は、C++ ファイルに対する有効な接尾辞として .cpp を追加します。次のように、makefile に SUFFIXES マクロを追加してください。

```
SUFFIXES:.cpp .cpp~
```

(この行は、makefile 内のどこにでも入れることができます。)

次の内容を makefile に追加します。インデントされている行は、必ずタブでインデントしてください。

```
.cpp:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.cpp~:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(LINK.cc) -o $@ $*.cpp $(LDLIBS)
.cpp.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp~.o:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp.a:
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
.cpp~.a:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
```

標準ライブラリヘッダーファイルに対する make の使用

標準ライブラリファイル名には、.h 接尾辞が付きません。標準ライブラリファイルには、istream、fstream、というような名前が付きます。また、テンプレートのソースファイルは、istream.cc、fstream.cc といった名前になります。

このため、Solaris 2.6 または Solaris 7 オペレーティング環境では、<istream> などの標準のライブラリヘッダーがプログラムにインクルードされていて、makefile に .KEEP_STATE がある場合は問題になります。たとえば <istream> を組み込むと、make ユーティリティは istream が実行可能ファイルであると思い、デフォルト規則を使用してistream.cc を元に istream を構築するので、非常に語弊のあるエラーメッセージができあがります(istream と istream.cc は、いずれも C++ インクルードファイルディレクトリにインストールされています)。1 つの解決策としては、make ユーティリティを使用せずに、dmake をシリアルモードで使用します (つまり、(dmake -m serial を実行)。また、当面の回避策としては、make に -r オプションを指定します。-r オプションはデフォルトの make 規則を無効にします。しかし、この解決策は構築プロセスまで破壊する可能性があります。第 3 の解決策は、.KEEP_STATE ターゲットを使用しないことです。

第3章

C++ コンパイラオプションの使い方

この章では、コマンド行 C++ コンパイラオプションの使用方法について説明してから、機能別にその使用方法を要約します。オプションの詳細は、付録 Aで説明します。

構文

次の表は、一般的なオプション構文の形式の例です。

表 3-1　オプション構文形式の例

構文形式	例
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

大括弧、括弧、中括弧、パイプ文字、および省略符号は、オプションの説明で使用されているメタキャラクタです。これらは、オプションの一部ではありません。構文の説明での表記規則は 本書の最初の「はじめに」を参照してください。

一般的な注意事項

C++ コンパイラのオプションを使用する際の一般的な注意事項は次のとおりです。

- `-l`*lib* オプションは、ライブラリ `lib`*lib*`.a` (または `lib`*lib*`.so`) とリンクするときに使用します。ライブラリが正しい順序で検索されるように、`-l`*lib* オプションは、ソースやオブジェクトのファイル名の後に指定する方が安全です。
- 一般にコンパイラオプションは左から右に処理され、マクロオプション (他のオプションを含むオプション) は条件に応じて内容が変更されます (ただし`-U` オプションだけは、すべての `-D` オプションの後に処理されます)。これはリンカーオプションには当てはまりません。
- `features`、`-I` `-l`、`-L`、`-library`、`-pti`、`-R`、`-staticlib`、`-U`、`-verbose` および `-xprefetch` オプションで指定した内容は蓄積され、上書きはされません。
- `D` オプションは累積されますが、同じ名前に複数の `-D` オプションがあるとお互いに上書きされます。

ソースファイル、オブジェクトファイル、ライブラリは、コマンド行に指定された順序でコンパイルおよびリンクされます。

機能別に見たオプションの要約

この節には、参照しやすいように、コンパイラオプションが機能別に分類されています。各オプションの詳細は、付録 Aを参照してください。

これらのオプションは、特に記載がない限りすべてのプラットフォームに適用されます。Solaris SPARC プラットフォーム版のオペレーティング環境に特有の機能は SPARC として表記され、Solaris Intel プラットフォーム版のオペレーティング環境に特有の機能は IA として表記されます。

コード生成オプション

コード生成オプションの要約をアルファベット順に示します。

表 3-2　コード生成オプション

オプション	処理
`-compat`	コンパイラの主要リリースとの互換モードを設定する。
`+e{0|1}`	仮想テーブル生成を制御する。
`-g`	デバッグ用にコンパルする。
`-KPIC`	位置に依存しないコードを生成する。
`-Kpic`	位置に依存しないコードを生成する。
`-mt`	マルチスレッド化したコードのコンパイルとリンクを行う。
`-xcode=`*a*	コードのアドレス空間を指定する。
`-xMerge`	データセグメントとテキストセグメントをマージする。
`+w`	意図しない結果が生じる可能性のあるコードを特定する。
`+w2`	`+w` で生成される警告以外に、通常は問題がなくても、プログラムの移植性を低下させる可能性がある技術的な違反についての警告も生成する。
`-xregs`	コンパイラは、一時記憶領域として使用できるレジスタ (一時レジスタ) が多ければ、それだけ高速なコードを生成します。このオプションは、利用できる一時レジスタを増やしますが、必ずしもそれが適切であるとは限りません。
`-z` *arg*	リンカーオプション

コンパイル時パフォーマンスオプション

コンパイル時パフォーマンスオプションの要約をアルファベット順に示します。

表 3-3 コンパイル時パフォーマンスオプション

オプション	処理
`-instlib`	指定ライブラリにすでに存在しているテンプレートインスタンスの生成を禁止する。
`-xjobs`	コンパイラが処理を行うために作成するプロセスの数を設定する。
`-xpch`	共通の一連のインクルードファイル群を共有するソースファイルを持つアプリケーションのコンパイル時間を短縮できる。
`-xpchstop`	`-xpch` でプリコンパイル済みヘッダーファイルを作成する際に適用される、最後のインクルードファイルを指定する。
`-xprofile_ircache`	`-xprofile=collect`*(SPARC)* で保存されたコンパイルデータを再使用する。
`-xprofile_pathmap`	*(SPARC)* 1 つのプロファイルディレクトリに存在する複数のプログラムや共有ライブラリのサポート。

デバッグオプション

デバッグオプションの要約をアルファベット順に示します。

表 3-4 デバッグオプション

オプション	処理
`+d`	C++ インライン関数を展開しない。
`-dryrun`	ドライバがコンパイラに渡すオプションを表示するが、コンパイルはしない。
`-E`	C++ ソースファイルにプリプロセッサを実行し、結果を `stdout` に出力するが、コンパイルはしない。
`-g`	デバッグ用にコンパルする。
`-g0`	デバッグ用にコンパイルするが、インライン機能は無効にしない。
`-H`	インクルードしたファイルのパス名を印刷する。

表 3-4　デバッグオプション (続き)

オプション	処理
-keeptmp	コンパイル中に作成されたすべての一時ファイルを残しておく。
-migration	以前のリソースからの移行に関する情報の参照先を表示する。
-P	ソースの前処理だけを行い、.i ファイルに出力する。
-Qoption	オプションをコンパイル中の各処理に直接渡す。
-readme	README ファイルの内容を表示する。
-s	実行可能ファイルからシンボルテーブルを取り除く。
-temp=*dir*	一時ファイルのディレクトリを指定する。
-verbose=*vlst*	コンパイラメッセージの詳細度を制御する。
-xcheck	スタックオーバーフローの実行時検査を追加する。
-xdumpmacros	定義内容、定義および解除された位置、使用されている場所に関する情報を出力する。
-xe	構文と意味のエラーのチェックだけを行う。
-xhelp=flags	コンパイラオプションの要約を一覧表示する。
-xildoff	インクリメンタルリンカーを無効にする。
-xildon	インクリメンタルリンカーを有効にする。
-xport64	32 ビットアーキテクチャから 64 ビットアーキテクチャへの移植中の一般障害について警告する。
-xs	オブジェクト (.o) ファイルなしに dbx でデバッグできるようにする。
-xsb	WorkShop ソースコードブラウザ用のテーブル情報を作成する。
-xsbfast	ソースブラウザ情報を作成するだけでコンパイルはしない。

浮動小数点オプション

浮動小数点オプションの要約をアルファベット順に示します。

表 3-5　浮動小数点オプション

オプション	処理
`-fns[={no|yes}]`	SPARC 非標準浮動小数点モードを有効または無効にする。
`-fprecision=`*p*	IA:浮動小数点精度モードを設定する。
`-fround=`*r*	起動時に IEEE 丸めモードを有効にする。
`-fsimple=`*n*	浮動小数点最適化の設定を行う。
`-fstore`	IA:浮動小数点式の精度を強制的に使用する。
`-ftrap=`*tlst*	起動時に IEEE トラップモードを有効にする。
`-nofstore`	IA:強制された式の精度を無効にする。
`-xlibmieee`	例外時に `libm` が数学ルーチンに対し IEEE 754 の値を返す。

言語オプション

言語オプションの要約をアルファベット順に示します。

表 3-6　言語オプション

オプション	処理
`-compat`	コンパイラの主要リリースとの互換モードを設定する。
`-features=`*alst*	C++ の各機能を有効化または無効化する。.
`-xchar`	文字型が符号なしと定義されているシステムからのコードの移行を容易に行えるようにする。
`-xldscope`	共有ライブラリをより速くより安全に作成するため、変数と関数の定義のデフォルトリンカースコープを制御する。
`-xthreadvar`	*(SPARC)*デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更する。
`-xtrigraphs`	文字表記シーケンスを認識する。
`-xustr`	16 ビット文字で構成された文字リテラルを認識する。

ライブラリオプション

ライブラリリンクオプションの要約をアルファベット順に示します。

表 3-7　ライブラリオプション

オプション	処理
-B*binding*	ライブラリのリンク形式を、シンボリック、動的、静的のいずれかから指定する。
-d{y|n}	実行可能ファイル全体に対し動的ライブラリを使用できるかどうか指定する。
-G	実行可能ファイルではなく動的共有ライブラリを構築する。
-h*name*	生成される動的共有ライブラリに名前を割り当てる。
-i	ld(1) がどのような LD_LIBRARY_PATH 設定も無視する。
-L*dir*	dir に指定したディレクトリを、ライブラリの検索に使用するディレクトリとして追加する。
-l*lib*	リンカーのライブラリ検索リストに liblib. または liblib.so を追加する。
-library=*llst*	特定のライブラリとそれに対応するファイルをコンパイルとリンクに強制的に組み込む。
-mt	マルチスレッド化したコードのコンパイルとリンクを行う。
-norunpath	ライブラリのパスを実行可能ファイルに組み込まない。
-R*plst*	動的ライブラリの検索パスを実行可能ファイルに組み込む。
-staticlib=*llst*	静的にリンクする C++ ライブラリを指定する。
-xar	アーカイブライブラリを作成する。
-xbuiltin[=*opt*]	標準ライブラリ呼び出しの最適化を有効または無効にする。
-xia	区間演算ライブラリをリンクし、適切な浮動小数点環境を設定する。
-xlang=*l*[,*l*]	該当する実行時ライブラリをインクルードし、指定された言語に適切な実行時環境を用意する。
-xlibmieee	例外時に libm が数学ルーチンに対し IEEE 754 の値を返す。
-xlibmil	最適化のために、選択された libm ライブラリルーチンをインライン展開する。
-xlibmopt	最適化された数学ルーチンを使用する。

表 3-7　ライブラリオプション (続き)

オプション	処理
-xlic_lib=sunperf	*(SPARC)* Sun Performance Library™ とリンクする。C++ の場合、-library=sunperf は、このライブラリをリンクするために適した方法である。
-xnativeconnect	オブジェクトファイルと以降の共有ライブラリ内にインタフェース情報を含めることで、共有ライブラリが Java™ プログラミング言語のコードとインタフェースをとれるようにする。
-xnolib	デフォルトのシステムライブラリとのリンクを無効にする。
-xnolibmil	コマンド行の -xlibmil を取り消す。
-xnolibmopt	数学ルーチンライブラリを使用しない。

ライセンスオプション

ライセンスオプションの要約をアルファベット順に示します。

表 3-8　ライセンスオプション

オプション	処理
-xlic_lib=sunperf	*(SPARC)* Sun Performance Library™ とリンクする。C++ の場合、-library=sunperf は、このライブラリをリンクするために適した方法である。
-xlicinfo	ライセンスサーバー情報を表示する。

廃止オプション

次のオプションはすでに廃止されているか、将来廃止されます。

表 3-9　廃止オプション

オプション	処理
-library=%all	将来のリリースで削除される。

表 3-9　廃止オプション

オプション	処理
-noqueue	ライセンス情報のキューイングを行わない
-ptr	コンパイラは無視する。将来のリリースのコンパイラがこのオプションを別の意味で使用する可能性もある。
-vdelx	将来のリリースで削除される。

出力オプション

次に、出力オプションについてアルファベット順に要約します。

表 3-10　出力オプション

オプション	処理
-c	コンパイルのみ。オブジェクト (.o) ファイルを作成するが、リンクはしない。
-dryrun	ドライバがコンパイラに渡すオプションを表示するが、コンパイルはしない。
-E	C++ ソースファイルにプリプロセッサを実行し、結果を stdout に出力するが、コンパイルはしない。
-erroff	コンパイラの警告メッセージを抑止する。
-errtags	各警告メッセージのメッセージタグを表示する。
-errwarn	指定の警告メッセージが出力されると、cc はエラーステータスを返して終了する。
-filt	コンパイラがリンカーエラーメッセージに適用するフィルタリングを抑止する。
-G	実行可能ファイルではなく動的共有ライブラリを構築する。
-H	インクルードしたファイルのパス名を印刷する。
-migration	以前のリソースからの移行に関する情報の参照先を表示する。
-o *filename*	出力ファイルや実行可能ファイルの名前を *filename* にする。
-P	ソースの前処理だけを行い、.i ファイルに出力する。
-Qproduce *sourcetype*	CC ドライバに *sourcetype* (ソースタイプ) 型のソースコードを生成するよう指示する。
-s	実行可能ファイルからシンボルテーブルを取り除く。

表 3-10 出力オプション (続き)

オプション	処理
-verbose=*vlst*	コンパイラメッセージの詳細度を制御する。
+w	必要に応じて追加の警告を出力する。
-w	警告メッセージを抑止する。
-xdumpmacros	定義内容、定義および解除された位置、使用されている場所に関する情報を出力する。
-xe	ソースファイルの構文と意味のチェックだけを行い、オブジェクトや実行可能コードの出力はしない。
-xhelp=flags	コンパイラオプションの要約を一覧表示する。
-xhelp=readme	README ファイルの内容を表示する。
-xM	makefile の依存情報を出力する。
-xM1	依存情報の生成は行うが、/usr/include の組み込みはしない。
-xsb	ソースコードブラウザ用のテーブル情報を作成する。
-xsbfast	ソースブラウザ情報を作成するだけでコンパイルはしない。
-xtime	コンパイル処理ごとの実行時間を報告する。
-xwe	ゼロ以外の終了状態を返すことによって、すべての警告をエラーに変換する。
-z *arg*	リンカーオプション

実行時パフォーマンスオプション

実行時パフォーマンスオプションの要約をアルファベット順に示します。

表 3-11 実行時パフォーマンスオプション

オプション	処理
-fast	一部のプログラムで最適な実行速度が得られるコンパイルオプションの組み合わせを選択する。
-g	パフォーマンスの解析 (およびデバッグ) に備えてプログラムを用意するようにコンパイラとリンカーの両方に指示する。
-s	実行可能ファイルからシンボルテーブルを取り除く。

表 3-11 実行時パフォーマンスオプション (続き)

オプション	処理
-xalias_level	コンパイラで、型に基づく別名の解析および最適化を実行するように指定する。
-xarch=*isa*	ターゲットのアーキテクチャ命令セットを指定する。
-xbuiltin[=*opt*]	標準ライブラリ呼び出しの最適化を有効または無効にする。
-xcache=*c*	*(SPARC)* オプティマイザのターゲットキャッシュ属性を定義する。
-xcg89	一般的な SPARC アーキテクチャ用のコンパイルを行う。
-xcg92	SPARC V8 アーキテクチャ用のコンパイルを行う。
-xchip=*c*	ターゲットのプロセッサチップを指定する。
-xF	リンカーによる関数と変数の順序変更を有効にする。
-xinline=*flst*	どのユーザーが作成したルーチンをオプティマイザでインライン化するかを指定する。
-xipo	内部手続きの最適化を実行する。
-xlibmil	最適化のために、選択された libm ライブラリルーチンをインライン展開する。
-xlibmopt	最適化された数学ルーチンライブラリを使用する。
-xlinkopt	*(SPARC)* オブジェクトファイル内のあらゆる最適化のほかに、結果として出力される実行可能ファイルや動的ライブラリのリンク時最適化も行う。
-xmemalign=*ab*	予想される最大メモリー整列と非整列データアクセスの動作を指定する。
-xnolibmil	コマンド行の -xlibmil を取り消す。
-xnolibmopt	数学ルーチンライブラリを使用しない。
-x*Olevel*	最適化レベルを level にする。
-xpagesize	スタックとヒープの優先ページサイズを設定する。
-xpagesize_heap	ヒープの優先ページサイズを設定する。
-xpagesize_stack	スタックの優先ページサイズを設定する。
-xprefetch[=*lst*]	*(SPARC)* 先読みをサポートするアーキテクチャーで先読み命令を有効にする。

表 3-11 実行時パフォーマンスオプション (続き)

オプション	処理
-xprefetch_level	-xprefetch=auto を設定したときの先読み命令の自動挿入を制御する。
-xprofile	*(SPARC)*実行時プロファイルデータを収集したり、このデータを使って最適化する。
-xregs=*rlst*	*(SPARC)* 一時レジスタの使用を制御する。
-xsafe=mem	*(SPARC)*メモリーに関するトラップを起こさないものとする。
-xspace	*(SPARC)*コードサイズが大きくなるような最適化は行わない。
-xtarget=*t*	ターゲットの命令セットと最適化のシステムを指定する。
-xthreadvar	*(SPARC)*デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更する。
-xunroll=*n*	可能な場合は、ループを展開する。
-xvis	*(SPARC)* VIS™ 命令セットに定義されているアセンブリ言語テンプレートをコンパイラが認識する。

プリプロセッサオプション

プリプロセッサオプションの要約をアルファベット順に示します。

表 3-12 プリプロセッサオプション

オプション	処理
-D*name*[=*def*]	シンボル *name* をプリプロセッサに定義する。
-E	C++ ソースファイルにプリプロセッサを実行し、結果を stdout に出力するが、コンパイルはしない。
-H	インクルードしたファイルのパス名を印刷する。
-P	ソースの前処理だけを行い、.i ファイルに出力する。
-U*name*	プリプロセッサシンボル *name* の初期定義を削除する。
-xM	makefile の依存情報を出力する。
-xM1	依存情報を生成するが、/usr/include は除く。

プロファイルオプション

プロファイルオプションの要約についてアルファベット順に示します。

表 3-13 プロファイルオプション

オプション	処理
`-p`	`prof` でプロファイル処理するためのデータを収集するオブジェクトコードを用意する。
`-xa`	プロファイル用のコードを生成する。
`-xpg`	`gprof` プロファイラによるプロファイル処理用にコンパイルする。
`-xprofile`	*(SPARC)*実行時プロファイルデータを収集したり、このデータを使って最適化する。

リファレンスオプション

次のオプションはコンパイラ情報を簡単に参照するためのものです。

表 3-14 リファレンスオプション

オプション	処理
`-migration`	以前のリソースからの移行に関する情報の参照先を表示する。
`-xhelp=flags`	コンパイラオプションの要約を一覧表示する。
`-xhelp=readme`	`README` ファイルの内容を表示する。

ソースオプション

ソースオプションの要約をアルファベット順に示します。

表 3-15 ソースオプション

オプション	処理
`-H`	インクルードしたファイルのパス名を印刷する。
`-I`*pathname*	`include` ファイル検索パスに*pathname* を追加する。

表 3-15 ソースオプション

オプション	処理
-I-	インクルードファイル検索規則を変更する。
-xM	makefile の依存情報を出力する。
-xM1	依存情報を生成するが、/usr/include は除く。

テンプレートオプション

テンプレートオプションの要約をアルファベット順に示します。

表 3-16 テンプレートオプション

オプション	処理
-instances=*a*	テンプレートインスタンスの位置とリンケージを制御する。
-pti*path*	テンプレートソースの検索ディレクトリを追加指定する。
-template=*wlst*	さまざまなテンプレートオプションを有効または無効にする。

スレッドオプション

スレッドオプションの要約をアルファベット順に示します。

表 3-17 スレッドオプション

オプション	処理
-mt	マルチスレッド化したコードのコンパイルとリンクを行う。
-xsafe=mem	*(SPARC)*メモリーに関するトラップを起こさないものとする。
-xthreadvar	*(SPARC)*デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更する。

PART II C++ プログラムの作成

第4章

言語拡張

この章では、このコンパイラ特有の言語拡張について説明します。付録 B にも、実装別情報を記載してあります。この章で扱っている機能のなかには、コマンド行でコンパイラオプションを指定しないかぎり、コンパイラが認識しないものがあります。関連するコンパイラオプションは、各セクションに適宜記載します。

`-features=extensions` オプションを使用すると、他の C++ コンパイラで一般的に認められている非標準コードをコンパイルすることができます。このオプションは、不正なコードをコンパイルする必要があり、そのコードを変更することが認められていない場合に使用することができます。

この章では、`-features=extensions` オプションを使用した場合にサポートされる言語拡張について説明します。

注 – 不正なコードは、どのコンパイラでも受け入れられる有効なコードに簡単に変更することができます。コードの変更が認められている場合は、このオプションを使用する代わりに、コードを有効なものに変更してください。
`-features=extensions` オプションを使用すると、コンパイラによっては受け入れられない不正なコードが残ることになります。

リンカースコープ

次の宣言指定子を、外部シンボルの宣言や定義の隠蔽のために使用します。静的なアーカイブやオブジェクトファイルに対して指定したスコープは、共有ライブラリや実行可能ファイルにリンクされるまで、適用されません。しかしながら、コンパイラは、与えられたリンカースコープ指定子に応じたいくつかの最適化を行うことができます。

これらの指定子を使用することで、リンカースコープのためにマップファイルを使う必要がなくなります。`-xldscope` をコマンド行で指定することによって、変数スコープのデフォルト設定を制御することもできます。

詳細は、360 ページの「`-xldscope={v}`」を参照してください。

表 4-1　宣言指定子

値	内容
`__global`	シンボル定義には大域リンカースコープとなります。これは、もっとも限定的でないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープが、外部シンボルのデフォルトのリンカースコープです。
`__symbolic`	シンボル定義は、シンボリックリンカースコープとなります。これは、大域リンカースコープより限定的なリンカースコープです。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュール外については、シンボルは大域なものとなります。このリンカースコープは、リンカーオプション `-Bsymbolic` に対応しています。C++ ライブラリでは `-Bsymbolic` を使用できませんが、`__symbolic` 指定子は問題なく使用できます。リンカーの詳細については、`ld`(1) を参照してください。
`__hidden`	シンボル定義は、隠蔽リンカースコープとなります。隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも限定的なリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。モジュールの外からは、シンボルは見えません。

より限定的な指定子を使ってシンボル定義を宣言しなおすことはできますが、より限定的でない指定子を使って宣言しなおすことはできません。いったん定義したシンボルを別の指定子によって宣言することはできません。

__global はもっとも限定的でないスコープ、__symbolic はそれよりも限定的なスコープであり、__hidden はもっとも限定的なスコープです。

仮想関数の宣言は仮想テーブルの構造と解釈に影響を及ぼすので、あらゆる仮想関数は、クラス定義を含んでいるあらゆるコンパイル単位から認識される必要があります。

C++ クラスでは、仮想テーブルや実行時型情報といった暗黙の情報の生成が必要となることがあるため、構造体、クラス、および共用体の宣言と定義にリンカースコープ指定子を適用できるようになっています。その場合、指定子は、構造体、クラス、または共用体キーワードの直後に置きます。こういったアプリケーションでは、すべての暗黙のメンバーに対して 1 つのリンカースコーピングが適用されます。

スレッドローカルな記憶装置

スレッドローカルな記憶装置を利用するには、スレッドローカルな変数を宣言します。スレッドローカルな変数の宣言は、通常の変数宣言に宣言指定子 __thread を加えたものです。詳細は、414 ページの「-xthreadvar[=o]」を参照してください。

__thread 指定子は、スレッド変数の最初の宣言部分に含める必要があります。

__thread 指定子で宣言できるのは、静的期間を持つ変数だけです。静的期間を持つ変数とは、ファイル内で大域なもの、ファイル内で静的なもの、関数内ローカルでかつ静的なもの、クラスの静的メンバなどが含まれます。期間が動的または自動である変数を __thread 指定子を使って宣言することは避けてください。スレッド変数に静的な初期設定子を持たせることはできますが、動的な初期設定子を持たせることはできません。たとえば `__thread int x = 4` を使用することはできますが、`__thread int x = f();` を使用することはできません。スレッド変数には、特殊なコンストラクタやデストラクタを持たせるべきではありません。とくに std::string 型をスレッド変数として持たせることはできません。

__thread 指定子で宣言した変数は、__thread 指定子がない場合と同じように結合されます。

スレッド変数の演算子 (&) のアドレスは、実行時に評価され、現在のスレッドの変数のアドレスが返されます。したがって、スレッド変数のアドレスは定数ではありません。結果として、スレッド変数のアドレスに設定されたとき、静的期間を持つ変数は動的に初期化されます。

スレッド変数のアドレスは、対応するスレッドの寿命のあいだ、安定しています。プロセス内のスレッドであればどれでも、変数の寿命期間中にスレッド変数のアドレスを自由に使用できます。スレッド終了後には、スレッド変数のアドレスを使用できません。スレッドの変数のアドレスは、スレッドが終了するとすべて無効となります。

例外の制限の少ない仮想関数による置き換え

C++ 標準では、関数を仮想関数で置き換える場合に、置き換える側の仮想関数で、置き換えられる側の関数より制限の少ない例外を指定することはできません。置き換える側の関数の例外指定は、置き換えられる側の関数と同じか、それよりも制限されていなければなりません。例外指定がないと、あらゆる例外が認められてしまうことに注意してください。

たとえば、基底クラスのポインタを使用して関数を呼び出す場合を考えてみましょう。その関数に例外指定が含まれていれば、それ以外の例外が送出されることはありません。しかし、置き換える側の関数で、それよりも制限の少ない例外指定が定義されている場合は、予期しない例外が送出される可能性があり、その結果としてプログラムが異常終了することがあります。これが、上で述べた規則がある理由です。

`-features=extensions` オプションを使用すると、限定の少ない例外指定を含んだ関数による置き換えが認められます。

enum 型と enum 変数の前方宣言

`features=extensions` オプションを使用すると、enum 型や enum 変数の前方宣言が認められます。さらに、不完全な enum 型による変数宣言も認められます。不完全な enum 型は、現行のプラットフォームの `int` 型と同じサイズと範囲を持つと想定されます。

次の 2 つの行は、-features=extensionsオプションを使用した場合にコンパイルされる不正なコードの例です。

```
enum E; // 不正:enum の前方宣言は認められていない
E e;    // 不正:型 E が不完全
```

enum 定義では、他の enum 定義を参照できず、他の型の相互参照もできないため、列挙型の前方宣言は必要ありません。コードを有効なものにするには、enum を使用する前に、その定義を完全なものにしておきます。

注 – 64 ビットアーキテクチャでは、enum のサイズを int よりも大きくしなければならない場合があります。その場合に、前方宣言と定義が同じコンパイルの中で見つかると、コンパイラエラーが発生します。実際のサイズが想定されたサイズと異なっていて、コンパイラがそのことを検出できない場合は、コードのコンパイルとリンクは行われますが、実際のプログラムが正しく動作する保証はありません。特に、8 バイト値が 4 バイト変数に格納されると、プログラムの動作が不正になる可能性があります。

不完全な enum 型の使用

-features=extensions オプションを使用した場合は、不完全な enum 型は前方宣言と見なされます。たとえば、このオプションを使用すると、次の不正なコードのコンパイルが可能になります。

```
typedef enum E F; // 不正: E が不完全
```

前述したように、enum 型を使用する前に、その定義を記述しておくことができます。

enum 名のスコープ修飾子としての使用

enum 宣言ではスコープを指定できないため、enum 名をスコープ修飾子として使用することはできません。たとえば、次のコードは不正です。

```
enum E { e1, e2, e3 };
int i = E::e1; // 不正:E はスコープ名ではない
```

この不正なコードをコンパイルするには、-features=extensionsオプションを使用します。このオプションを使用すると、スコープ修飾子が enum 型の名前だった場合に、その修飾子が無視されます。

このコードを有効なものにするには、不正な修飾子 E:: を取り除きます。

注 – このオプションを使用すると、プログラムのタイプミスが検出されずにコンパイルされる可能性が高くなります。

名前のない struct 宣言の使用

名前のない構造体宣言は、構造体のタグも、オブジェクト名も、typedef 名も指定されていない宣言です。C++ では、名前のない構造体は認められていません。

-features=extensions オプションを使用すると、名前のない struct 宣言を使用できるようになります。ただし、この宣言は共用体のメンバーとしてだけ使用することができます。

以下は、-features=extensions オプションを使用した場合にコンパイルが可能な、名前のない不正な struct 宣言の例です。

```
union U {
  struct {
    int a;
    double b;
  };  // 不正: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // 不正: anonymous struct
};
```

これらの構造体のメンバー名は、構造体名で修飾しなくても認識されます。たとえば、共用体 U が上のコードのように定義されているとすると、次のような記述が可能です。

```
U u;
u.a = 1;
```

名前のない構造体は、名前のない共用体と同じ制約を受けます。

コードを有効なものにするには、次のようにそれぞれの構造体に名前を付けます。

```
union U {
  struct {
    int a;
    double b;
  } A;
  struct {
    char* c;
    unsigned d;
  } B;
};
U u;
U.A.a = 1;
```

名前のないクラスインスタンスのアドレスの受け渡し

一時変数のアドレスは取得できません。たとえば、次のコードは不正です。コンストラクタ呼び出しによって作成された変数のアドレスが取得されてしまうからです。ただし、-features=extensions オプションを使用した場合は、この不正なコードもコンパイル可能になります。

```
class C {
  public:
    C(int);
    ...
};
void f1(C*);
int main()
{
  f1( &C(2) ); // 不正
}
```

このコードを有効なものにするには、次のように明示的な変数を使用します。

```
C c(2);
f1(&c);
```

一時オブジェクトは、関数が終了したときに破棄されます。一時変数のアドレスを取得しないようにするのは、プログラムの作成者の責任になります。また、(f1 などで)一時変数に格納されたデータは、その変数が破棄されたときに失われます。

静的名前空間スコープ関数のクラスフレンドとしての宣言

次のコードは不正です。

```
class A {
  friend static void foo(<args>);
  ...
};
```

クラス名に外部リンケージが含まれていて、それぞれの定義は別々でなければならないため、フレンド関数にも外部リンケージが含まれていなければなりません。しかし、-features=extensions オプションを使用すると、このコードもコンパイルできるようになります。

おそらく、この不正なコードの目的は、クラス A の実装ファイルに、メンバーではない「ヘルパー」関数を組み込むことでしょう。そうであれば、foo を静的メンバー関数にしても結果は同じです。クライアントから呼び出せないように、この関数を非公開にすることもできます。

注 – この拡張機能を使用すると、作成したクラスを任意のクライアントが「横取り」できるようになります。そのためには、任意のクライアントにこのクラスのヘッダーを組み込み、独自の静的関数 foo を定義します。この関数は、自動的にこのクラスのフレンド関数になります。その結果は、このクラスのメンバーをすべて公開にした場合と同じになります。

事前定義済み __func__ シンボルの関数名としての使用

-features=extensionsオプションを使用すると、それぞれの関数で __func__ 識別子が const char 型の静的配列として暗黙的に宣言されます。プログラムの中で、この識別子が使用されていると、コンパイラによって次の定義が追加されます。ここで、*function-name* は関数の単純名です。この名前には、クラスメンバーシップ、名前空間、多重定義の情報は反映されません。

```
static const char __func__[] = "function-name";
```

たとえば、次のコードを考えてみましょう。

```
#include <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);
}
```

この関数が呼び出されるたびに、標準出力ストリームに次の情報が出力されます。

```
myfunc
```

第5章

プログラムの編成

C++ プログラムのファイル編成は、C プログラムの場合よりも慎重に行う必要があります。この章では、ヘッダーファイルとテンプレート定義の設定方法について説明します。

ヘッダーファイル

有効なヘッダーファイルを簡単に作成できるとはかぎりません。場合によっては、C と C++ の複数のバージョンで使用可能なヘッダーファイルを作成する必要があります。また、テンプレートを使用するためには、複数回の包含 (べき等) が可能なヘッダーファイルが必要です。

言語に対応したヘッダーファイル

場合によっては、C と C++ の両方のプログラムにインクルード可能なヘッダーファイルを作成する必要があります。ただし、従来の C とも呼ばれる Kernighan & Ritchie C (K&R C) や、ANSI C、Annotated Reference Manual C++ (ARM C++)、および ISO C++ では、1 つのヘッダーファイル内の同一のプログラム要素について異なった宣言や定義が規定されていることがあります(異なる言語とバージョンについての詳細は『C++ 移行ガイド』を参照してください)。これらのどの標準言語でも同じヘッダーファイルを使用できるようにするには、プロセッサマクロ `__STDC__` や `__cplusplus` の定義の有無や、その値に基づく条件付きコンパイルを使用する必要があります。

__STDC__ マクロは、K&R C では定義されていませんが、ANSI C や C++ では定義されています。このマクロが定義されているかどうかを使用して、K&R C のコードを ANSI C や C++ のコードから区別します。このマクロは、プロトタイプの関数定義とプロトタイプではない関数定義を分離するときに特に役立ちます。

```
#ifdef __STDC__
int function(char*,...);      // C++ & ANSI C declaration
#else
int function();               // K&R C
#endif
```

__cplusplus マクロは、C では定義されていませんが、C++ では定義されています。

注 – 旧バージョンの C++ では、__cplusplus の代わりに c_plusplus マクロが定義されていました。c_plusplus マクロは、現在のバージョンでは定義されていません。

---

cplusplus マクロが定義されているかどうかを使用して、C と C++ を区別します。このマクロは、次のように関数宣言用の extern "C" インタフェースを保護するときに特に便利です。extern "C" の指定の一貫性を保つため、extern "C" のリンケージ指定のスコープ内には #include 指令を含めないでください。

```
#include "header.h"
...                       // ... other include files ...
#if defined(__cplusplus)
extern "C" {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

ARM C++ では、__cplusplus マクロの値が 1 になり、ISO C++ では、このマクロの値が 199711L (この規格の制定年月の long 定数表現) になります。この値の違いを使用して、ARM C++ と ISO C++ を区別します。これらのマクロ値は、テンプレート構文の違いを保護するときに特に役立ちます。

```
// テンプレート関数の指定
#if __cplusplus < 199711L
int power(int,int);                          // ARM C++
#else
template <> int power(int,int);              // ISO C++
#endif
```

べき等ヘッダーファイル

ヘッダーファイルはべき等にしてください。すなわち、同じヘッダーファイルを何回インクルードしても、1 回だけインクルードした場合と効果が同じになるようにしてください。このことは、テンプレートでは特に重要です。べき等を実現するもっともよい方法は、プリプロセッサの条件を設定し、ヘッダーファイルの本体の重複を防止することです。

```
#ifndef HEADER_H
#define HEADER_H
/* contents of header file */
#endif
```

テンプレート定義

テンプレート定義は 2 通りの方法で編成することができます。すなわち、テンプレート定義を取り込む方法 (定義取り込み型編成) と、分離する方法 (定義分離型編成) があります。テンプレート定義を取り込んだほうが、テンプレートのコンパイルを制御しやすくなります。

テンプレート定義の取り込み

定義取り込み型編成とは、テンプレートの宣言と定義を、そのテンプレートを使用するファイルの中に含めることです。構築の例を次に示します。

```
main.cc     template <class Number> Number twice( Number original
            );
            template <class Number> Number twice( Number original )
                { return original + original; }
            int main( )
                { return twice<int>( -3 ); }
```

テンプレートを使用するファイルに、テンプレートの宣言と定義の両方を含んだファイルをインクルードした場合も、定義取り込み型編成を使用したことになります。構築の例を次に示します。

```
twice.h     #ifndef TWICE_H
            #define TWICE_H
            template <class Number> Number twice( Number original
            );
            template <class Number> Number
            twice( Number original )
                { return original + original; }
            #endif

main.cc     #include "twice.h"
            int main( )
                { return twice( -3 ); }
```

注 – テンプレートのヘッダーは、べき等にすることが特に重要です (61 ページの「べき等ヘッダーファイル」を参照してください)。

テンプレート定義の分離

テンプレート定義を編成するもう一つの方法は、テンプレートの定義をテンプレート定義ファイルに記述することです。この例を次に示します。

```
twice.h     #ifndef TWICE_H
            #define TWICE_H
            template <class Number> Number twice
              ( Number original );
            #endif

twice.cc    template <class Number> Number twice
              ( Number original )
                { return original + original; }

main.cc     #include "twice.h"
            int main( )
                { return twice<int>( -3 ); }
```

テンプレート定義ファイルには、べき等ではないヘッダーファイルをインクルードしてはいけません。また、通常はテンプレート定義ファイルにヘッダーファイルをインクルードする必要はありません(61 ページの「べき等ヘッダーファイル」を参照してください)。なお、テンプレートの定義分離型編成は、すべてのコンパイラでサポートされているわけではありません。

独立した定義ファイルはヘッダーファイルなので、多数のファイルに暗黙のうちにインクルードされることがあります。そのため、テンプレート定義の一部でないかぎり、あらゆる関数と変数はこのファイルに含めないようにします。独立した定義ファイルには、typedef などの型定義を定義できます。

注 – 通常、テンプレート定義ファイルには、ソースファイルの拡張子 (`.c`、`.C`、`.cc`、`.cpp`、`.cxx`、`.c++`のいずれか) を付けますが、このファイルはヘッダーファイルです。コンパイラは、これらのファイルを必要に応じて自動的に取り込みます。テンプレート定義ファイルの単独コンパイルは行わないでください。

このように、テンプレートの宣言と定義を別々のファイルで指定した場合は、定義ファイルの内容、その名前、配置先に特に注意する必要があります。さらに、定義ファイルの配置先をコンパイラに明示的に通知する必要もあります。テンプレート定義の検索規則については、94 ページの「テンプレート定義の検索」を参照してください。

第6章

テンプレートの作成と使用

テンプレートの目的は、プログラマが一度コードを書くだけで、そのコードが型の形式に準拠して広範囲の型に適用できるようにすることです。この章では関数テンプレートに関連したテンプレートの概念と用語を紹介し、より複雑な (そして、より強力な) クラステンプレートと、テンプレートの使用方法について説明しています。また、テンプレートのインスタンス化、デフォルトのテンプレートパラメータ、およびテンプレートの特殊化についても説明しています。この章の最後には、テンプレートの潜在的な問題が挙げられています。

関数テンプレート

関数テンプレートは、引数または戻り値の型だけが異なった、関連する複数の関数を記述したものです。

関数テンプレートの宣言

テンプレートは使用する前に宣言しなければなりません。次の例に見られるように、「宣言」によってテンプレートを使用するのに十分な情報は与えられますが、テンプレートの実装には他の情報も必要です。

```
template <class Number> Number twice( Number original );
```

この例では *Number* は「テンプレートパラメータ」であり、テンプレートが記述する関数の範囲を指定します。つまり、*Number* は「テンプレート型のパラメータ」です。テンプレート定義内で使用すると、型はテンプレートを使用するときに特定されることになります。

関数テンプレートの定義

テンプレートは宣言と定義の両方が必要になります。テンプレートを「定義」することで実装に必要な情報が得られます。次の例は、前述の例で宣言されたテンプレートを定義しています。

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

テンプレート定義は通常ヘッダーファイルで行われるので、テンプレート定義が複数のコンパイル単位で繰り返される可能性があります。しかし、すべての定義は同じでなければなりません。この制限は「単一定義ルール」と呼ばれます。

コンパイラは、関数パラメータリスト内にテンプレートの型名でないパラメータを含む式をサポートしていません。次に例を示します。

```
// パラメータリスト内にテンプレートの型名でない
// テンプレートパラメータを持つ式はサポートされていません。
template<int I> void foo( mytype<2*I> ) { ... }
template<int I, int J> void foo( int a[I+J] ) { ... }
```

関数テンプレートの使用

テンプレートは、いったん宣言すると他のすべての関数と同様に使用することができます。テンプレートを「使用」するには、そのテンプレートの名前とテンプレート引数を指定します。コンパイラは、テンプレート型引数を、関数引数の型から推測します。たとえば、以前に宣言されたテンプレートを次のように使用できます。

```
double twicedouble( double item )
    { return twice( item ); }
```

テンプレート引数が関数の引数型から推測できない場合、その関数が呼び出される場所にその引数を指定する必要があります。例を次に示します。

```
template<class T> T func(); // 関数引数なし
int k = func<int>(); // テンプレート引数を明示的に指定
```

クラステンプレート

クラステンプレートは、複数の関連するクラス (データ型) を記述します。グラステンプレートに記述されているクラスは、型のほかに整数値、または大域リンケージによる変数へのポインタや参照だけが互いに異なっています。クラステンプレートは、一般的ではあるけれども型が保証されているデータ構造を記述するのに特に便利です。

クラステンプレートの宣言

クラステンプレートの宣言では、クラスの名前とそのテンプレート引数だけを指定します。このような宣言は「不完全なクラステンプレート」と呼ばれます。

次の例は、任意の型の引数をとる `Array` というクラスに対するテンプレート宣言の例です。

```
template <class Elem> class Array;
```

次のテンプレートは、`unsigned int` の引数をとる `String` というクラスに対する宣言です。

```
template <unsigned Size> class String;
```

クラステンプレートの定義

クラステンプレートの定義では、次の例のようにクラスデータと関数メンバーを宣言しなければなりません。

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize();
        Elem& operator[]( int idx );
};
```

```
template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

関数テンプレートとは違って、クラステンプレートには `class Elem` のような型パラメータと `unsigned Size` のような式パラメータの両方を指定できます。式パラメータには次の情報を指定できます。

- 整数型または列挙型を持つ値
- オブジェクトへのポインタまたは参照
- 関数へのポインタまたは参照
- クラスメンバー関数へのポインタ

クラステンプレートメンバーの定義

クラステンプレートを完全に定義するには、その関数メンバーと静的データメンバーを定義する必要があります。動的 (静的でない) データメンバーの定義は、クラステンプレート宣言で十分です。

関数メンバーの定義

テンプレート関数メンバーの定義は、テンプレートパラメータの指定と、それに続く関数定義から構成されます。関数識別子は、クラステンプレートのクラス名とそのテンプレートの引数で修飾されます。次の例は、`template <class Elem>` というテンプレートパラメータ指定を持つ `Array` クラステンプレートの 2 つの関数メンバー定義を示しています。それぞれの関数識別子は、テンプレートクラス名とテンプレート引数 `Array<Elem>` で修飾されています。

```
template <class Elem> Array<Elem>::Array( int sz )
    { size = sz; data = new Elem[ size ]; }

template <class Elem> int Array<Elem>::GetSize( )
    { return size; }
```

次の例は、String クラステンプレートの関数メンバーの定義を示しています。

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    { int len = 0;
      while ( len < Size && data[len] != '\0' ) len++;
      return len; }

template <unsigned Size> String<Size>::String( char *initial )
    { strncpy( data, initial, Size );
      if ( length( ) == Size ) overflows++; }
```

静的データメンバーの定義

テンプレートの静的データメンバーの定義は、テンプレートパラメータの指定と、それに続く変数定義から構成されます。この場合、変数識別子は、クラステンプレート名とそのテンプレートの実引数で修飾されます。

```
template <unsigned Size> int String<Size>::overflows = 0;
```

クラステンプレートの使用

テンプレートクラスは、型が使用できる場所ならどこででも使用できます。テンプレートクラスを指定するには、テンプレート名と引数の値を設定します。次の宣言例では、Array テンプレートに基づいた変数 int_array を作成します。この変数のクラス宣言とその一連のメソッドは、Elem が int に置き換わっている点以外は、Array テンプレートとまったく同じです (70 ページの「テンプレートのインスタンス化」を参照)。

```
Array<int> int_array( 100 );
```

次の宣言例は、String テンプレートを使用して short_string 変数を作成します。

```
String<8> short_string( "hello" );
```

テンプレートクラスのメンバー関数は、他のすべてのメンバー関数と同じように使用できます。

```
int x = int_array.GetSize( );
```

```
int x = short_string.length( );
```

テンプレートのインスタンス化

テンプレートの「インスタンス化」には、特定の組み合わせのテンプレート引数に対応した具体的なクラスまたは関数 (「インスタンス」) を生成することが含まれます。たとえば、コンパイラは`Array<int>` と`Array<double>` に対応した別々のクラスを生成します。これらの新しいクラスの定義では、テンプレートクラスの定義の中のテンプレートパラメータがテンプレート引数に置き換えられます。前述の「クラステンプレート」の節に示す `Array<int>` の例では、すべての`Elem` が `int` に置き換えられます。

テンプレートの暗黙的インタンス化

テンプレート関数またはテンプレートクラスを使用すると、インスタンス化が必要になります。そのインスタンスがまだ存在していない場合には、コンパイラはテンプレート引数に対応したテンプレートを暗黙的にインスタンス化します。

テンプレートの明示的インスタンス化

コンパイラは、実際に使用されるテンプレート引数に対応したテンプレートだけを暗黙的にインスタンス化します。これは、テンプレートを持つライブラリの作成には適していない可能性があります。C++ には、次の例のように、テンプレートを明示的にインスタンス化するための手段が用意されています。

テンプレート関数の明示的インスタンス化

テンプレート関数を明示的にインスタンス化するには、template キーワードに続けて関数の宣言 (定義ではない) を行います。関数の宣言では関数識別子の後にテンプレート引数を指定します。

```
template float twice<float>( float original );
```

テンプレート引数は、コンパイラが推測できる場合は省略できます。

```
template int twice( int original );
```

テンプレートクラスの明示的インスタンス化

テンプレートクラスを明示的にインスタンス化するには、template キーワードに続けてクラスの宣言 (定義ではない) を行います。クラスの宣言ではクラス識別子の後にテンプレート引数を指定します。

```
template class Array<char>;
```

```
template class String<19>;
```

クラスを明示的にインスタンス化すると、そのメンバーもすべてインスタンス化されます。

テンプレートクラス関数メンバーの明示的インスタンス化

テンプレート関数メンバーを明示的にインスタンス化するには、template キーワードに続けて関数の宣言 (定義ではない) を行います。関数の宣言ではテンプレートクラスで修飾した関数識別子の後にテンプレート引数を指定します。

```
template int Array<char>::GetSize( );
```

```
template int String<19>::length( );
```

テンプレートクラスの静的データメンバーの明示的インスタンス化

テンプレートの静的データメンバーを明示的にインスタンス化するには、template キーワードに続けてメンバーの宣言 (定義ではない) を行います。メンバーの宣言では、テンプレートクラスで修飾したメンバー識別子の後にテンプレート引数を指定します。

```
template int String<19>::overflows;
```

テンプレートの編成

テンプレートは、入れ子にして使用できます。これは、標準 C++ ライブラリで行う場合のように、一般的なデータ構造に関する汎用関数を定義する場合に特に便利です。たとえば、テンプレート配列クラスに関して、テンプレートのソート関数を次のように宣言することができます。

```
template <class Elem> void sort( Array<Elem> );
```

また、次のように定義することができます。

```
template <class Elem> void sort( Array<Elem> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( store[j-1] > store[j] )
                  { Elem temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

前の例は、事前に宣言された Array クラステンプレートのオブジェクトに関するソート関数を定義しています。次の例はソート関数の実際の使用例を示しています。

```
Array<int> int_array( 100 );   // intの配列を作成し、
sort( int_array );         // それをソートする。
```

デフォルトのテンプレートパラメータ

クラステンプレートのテンプレートパラメータには、デフォルトの値を指定できます(関数テンプレートは不可)。

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

テンプレートパラメータにデフォルト値を指定する場合、それに続くパラメータもすべてデフォルト値でなければなりません。テンプレートパラメータに指定できるデフォルト値は 1 つです。

テンプレートの特殊化

次の `twice` の例のように、テンプレート引数を例外的に特定の形式で組み合わせると、パフォーマンスが大幅に改善されることがあります。あるいは、次の `sort.` の例のように、テンプレート記述がある引数の組み合わせに対して適用できないこともあります。テンプレートの特殊化によって、実際のテンプレート引数の特定の組み合わせに対して代替実装を定義することが可能になります。テンプレートの特殊化はデフォルトのインスタンス化を無効にします。

テンプレートの特殊化宣言

前述のようなテンプレート引数の組み合わせを使用するには、その前に特殊化を宣言しなければなりません。次の例は *twice* と *sort* の特殊化された実装を宣言しています。

```
template <> unsigned twice<unsigned>( unsigned original );
```

```
template <> sort<char*>( Array<char*> store );
```

コンパイラがテンプレート引数を明確に確認できる場合には、次の例のようにテンプレート引数を省略することができます。

```
template <> unsigned twice( unsigned original );
```

```
template <> sort( Array<char*> store );
```

テンプレートの特殊化定義

宣言するテンプレートの特殊化はすべて定義しなければなりません。次の例は、前の節で宣言された関数を定義しています。

```
template <> unsigned twice<unsigned>( unsigned original )
    { return original << 1; }
```

```
#include <string.h>
template <> void sort<char*>( Array<char*> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( strcmp( store[j-1], store[j] ) > 0 )
                  { char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

テンプレートの特殊化の使用とインスタンス化

特殊化されたテンプレートは他のすべてのテンプレートと同様に使用され、インスタンス化されます。ただし、完全に特殊化されたテンプレートの定義はインスタンス化でもあります。

部分特殊化

前の例では、テンプレートは完全に特殊化されています。つまり、このようなテンプレートは特定のテンプレート引数に対する実装を定義しています。テンプレートは部分的に特殊化することも可能です。これは、テンプレートパラメータの一部だけを指定する、または、1 つまたは複数のパラメータを特定のカテゴリの型に制限すること

を意味します。部分特殊化の結果、それ自身はまだテンプレートのままです。たとえば、次のコード例に、本来のテンプレートとそのテンプレートの完全特殊化を示します。

```
template<class T, class U> class A { ... }; //本来のテンプレート
template<> class A<int, double> { ... };    //特殊化
```

次のコード例に、本来のテンプレートの部分特殊化を示します。

```
template<classU> class A<int> { ... };          // 例 1
template<class T, class U> class A<T*> { ... }; // 例 2
template<class T> class A<T**, char> { ... };   // 例 3
```

- 例 1 は、最初のテンプレートパラメータが int 型である特殊なテンプレート定義です。
- 例 2 は、最初のテンプレートパラメータが任意のポインタ型である、特殊なテンプレート定義です。
- 例 3 は、最初のテンプレートパラメータが任意の型のポインタへのポインタであり、2 番目のテンプレートパラメータが char 型である、特殊なテンプレート定義です。

テンプレートの問題

この節では、テンプレートを使用する場合の問題について説明しています。

非局所型名前の解決とインスタンス化

テンプレート定義で使用される名前の中には、テンプレート引数によって、またはそのテンプレート内で、定義されていないものがある可能性があります。そのような場合にはコンパイラが、定義の時点で、またはインスタンス化の時点で、テンプレートを取り囲むスコープから名前を解決します。1 つの名前が複数の場所で異なる意味を持つために解決の形式が異なることも考えられます。

名前の解決は複雑です。したがって、汎用性の高い標準的な環境で提供されているもの以外は、非局所型名前に依存することは避ける必要があります。言い換えれば、どこでも同じように宣言され、定義されている非局所型名前だけを使用するようにしてください。この例では、テンプレート関数の converter が、非局所型名前である intermediary と temporary を使用しています。これらの名前は use1.cc と use2.cc では異なる定義を持っているため、コンパイラが異なれば結果は違うものになるでしょう。テンプレートが正しく機能するためには、すべての非局所型名前 (intermediary と temporary) がどこでも同じ定義を持つ必要があります。

```
use_common.h     // 共通のテンプレート定義
                 template <class Source, class Target>
                 Target converter( Source source )
                        { temporary = (intermediary)source;
                        return (Target)temporary; }

use1.cc          typedef int intermediary;
                 int temporary;

                 #include "use_common.h"

use2.cc          typedef double intermediary;
                 unsigned int temporary;

                 #include "use_common.h"
```

非局所型名前を使用する典型的な例として、1 つのテンプレート内で cin と cout のストリームの使用があります。ほとんどのプログラマは実際、ストリームをテンプレートパラメータとして渡すことは望まないので、1 つの大域変数を参照するようにします。しかし、cin と cout はどこでも同じ定義を持っていなければなりません。

テンプレート引数としての局所型

テンプレートインスタンス化の際には、型と名前が一致することを目安に、どのテンプレートがインスタンス化または再インスタンス化される必要があるか決定されます。したがって、局所型がテンプレート引数として使用された場合には重大な問題が発生する可能性があります。自分のコードに同様の問題が生じないように注意してください。例を次に示します。

コード例 6-1 テンプレート引数としての局所型の問題の例

```
array.h     template <class Type> class Array {
                Type* data;
                int   size;
            public:
                Array( int sz );
                int GetSize( );
            };

array.cc    template <class Type> Array<Type>::Array( int sz )
                { size = sz; data = new Type[size]; }
            template <class Type> int Array<Type>::GetSize( )
                { return size;}

file1.cc    #include "array.h"
            struct Foo { int data; };
            Array<Foo> File1Data(10);

file2.cc    #include "array.h"
            struct Foo { double data; };
            Array<Foo> File2Data(20);
```

file1.cc の中に登録された Foo 型は、file2.cc の中に登録された Foo 型と同じではありません。局所型をこのように使用すると、エラーと予期しない結果が発生することがあります。

テンプレート関数のフレンド宣言

テンプレートは、使用前に宣言されていなければなりません。フレンド宣言では、テンプレートを宣言するのではなく、テンプレートの使用を宣言します。フレンド宣言の前に、実際のテンプレートが宣言されていなければなりません。次の例では、作成

済みオブジェクトファイルをリンクしようとするときに、operator<< 関数が未定義であるというエラーが生成されます。その結果、operator<< 関数はインスタンス化されません。

コード例 6-2 フレンド宣言の問題の例

```
array.h     // operator<< 関数に対して未定義エラーを生成する
            #ifndef ARRAY_H
            #define ARRAY_H
            #include <iosfwd>

            template<class T> class array {
                int size;
            public:
                array();
                friend std::ostream&
                    operator<<(std::ostream&, const array<T>&);
            };
            #endif

array.cc    #include <stdlib.h>
            #include <iostream>

            template<class T> array<T>::array() { size = 1024; }

            template<class T>
            std::ostream&
            operator<<(std::ostream& out, const array<T>& rhs)
                { return out << '[' << rhs.size << ']'; }

main.cc     #include <iostream>
            #include "array.h"

            int main()
            {
                std::cout
                  << "creating an array of int... " << std::flush;
                array<int> foo;
                std::cout << "done\n";
                std::cout << foo << std::endl;
                return 0;
            }
```

コンパイラは、次の宣言をarray クラスの friend である正規関数の宣言として読み取っているので、コンパイル中にエラーメッセージを表示しません。

```
friend ostream& operator<<(ostream&, const array<T>&);
```

operator<< は実際にはテンプレート関数であるため、template class array を宣言する前にこの関数にテンプレート宣言を行う必要があります。しかし、operator<< はパラメータtype array<T> を持つため、関数宣言の前に array<T> を宣言する必要があります。ファイル array.h は、次のようになります。

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// 次の 2 行は operator<< をテンプレート関数として宣言する
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

テンプレート定義内での修飾名の使用

C++ 標準は、テンプレート引数に依存する修飾名を持つ型を、typename キーワードを使用して型名として明示的に示すことを規定しています。これは、それが型であることをコンパイラが認識できる場合も同様です。次の例の各コメントは、それぞれの修飾名が typename キーワードを必要とするかどうかを示しています。

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // 型ではない
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;      // 型ではない
template <class T> struct example {
  static typename T::a_type variable1;              // 必要
  static typename parametric<T>::a_type variable2; // 必要
  static simple::a_type variable3;                   // 不要
};
template <class T> typename T::a_type                   // 必要
  example<T>::variable1 = 0;          // 型ではない
template <class T> typename parametric<T>::a_type      // 必要
  example<T>::variable2 = 0;          // 型ではない
template <class T> simple::a_type                       // 不要
example<T>::variable3 = 0;          // 型ではない
```

テンプレート宣言の入れ子

">>" という文字を持つものは右シフト演算子と解釈されるため、あるテンプレート宣言を別のテンプレート宣言内で使用する場合は注意が必要です。隣接する「>」文字との間に、少なくとも 1 つの空白文字を入れるようにしてください。

以下に誤った書式の例を示します。

```
//誤った書式の文
Array<String<10>> short_string_array(100); // >> は右シフトを示す。
```

上記の文は、次のように解釈されます。

```
Array<String<10  >> short_string_array(100);
```

正しい構文は次のとおりです。

```
Array<String<10> > short_string_array(100);
```

静的変数や静的関数の参照

テンプレート定義の内部では、大域スコープや名前空間で静的として宣言されたオブジェクトや関数の参照がサポートされません。複数のインスタンスが生成されると、それぞれのインスタンスが別々のオブジェクトを参照するため、一定義規約 (C++ 標準の第 3.2 節) に違反するためです。通常、このエラーはリンク時にシンボルの不足の形で通知されます。

すべてのテンプレートのインスタンス化で同じオブジェクトを共有させたい場合は、そのオブジェクトを該当する名前空間の非静的メンバーにします。また、あるテンプレートクラスをインスタンス化するたびに、別々のオブジェクトを使用したい場合は、そのオブジェクトを該当するテンプレートクラスの静的メンバーにします。同様に、あるテンプレート関数をインスタンス化するたびに、別々のオブジェクトを使用したい場合は、そのオブジェクトを該当するテンプレート関数の局所メンバーにします。

テンプレートを使用して複数のプログラムを同一ディレクトリに構築する

テンプレートを使用して複数のプログラムを構築する場合は、それらを別のディレクトリに構築することを推奨します。同一ディレクトリ内に構築する場合は、構築ごとにリポジトリを消去する必要があります。これにより、予期しないエラーが回避されます。詳細については、93 ページの「テンプレートリポジトリの共有」を参照してください。

makefile a.cc、b.cc、x.h、x.cc を使用した例で説明します。この例が意味を持つのは、-instances=extern を指定した場合だけです。

```
........
Makefile
........
CCC = CC

all: a b

a:
    $(CCC) -I. -instances=extern -c  a.cc
    $(CCC) -instances=extern -o a a.o

b:
    $(CCC) -I. -instances=extern -c  b.cc
    $(CCC) -instances=extern -o b b.o

clean:
    /bin/rm -rf SunWS_cache *.o a b
```

```
...
x.h
...
template <class T> class X {
public:
  int open();
  int create();
  static int variable;
};
```

```
...
x.cc
...
template <class T> int X<T>::create() {
  return variable;
}

template <class T> int X<T>::open() {
  return variable ;
}

template <class T> int X<T>::variable = 1;
```

```
...
a.cc
...
#include "x.h"

main()
{
  X<int> temp1;

  temp1.open();
  temp1.create();
}
```

```
...
b.cc
...
#include "x.h"

main()
{
  X<int> temp1;

  temp1.create();
}
```

a と b の両方を構築する場合は、それらの構築の間に make clean を実行します。以下のコマンドでは、エラーが発生します。

```
example% make a
example% make b
```

以下のコマンドでは、エラーは発生しません。

```
example% make a
example% make clean
example% make b
```

第7章

テンプレートのコンパイル

テンプレートをコンパイルするためには、C++ コンパイラは従来の UNIX コンパイラよりも多くのことを行う必要があります。C++ コンパイラは、必要に応じてテンプレートインスタンスのオブジェクトコードを生成しなければなりません。コンパイラは、テンプレートリポジトリを使って、別々のコンパイル間でテンプレートインスタンスを共有することができます。また、テンプレートコンパイルのいくつかのオプションを使用できます。コンパイラは、別々のソースファイルにあるテンプレート定義を見つけ、テンプレートインスタンスと `main` コード行の整合性を維持しなければなりません。

冗長コンパイル

フラグ `-verbose=template` が指定されている場合は、テンプレートコンパイル作業中の重要なイベントがユーザーに通知されます。逆に、デフォルトの `-verbose=no%template` が指定されている場合は、通知されません。そのほかに、`+w` オプションを指定すると、テンプレートのインスタンス化が行われたときに問題になりそうな内容が通知される場合があります。

テンプレートのインスタンス化

テンプレートリポジトリの管理は CCadmin(1) コマンドで行います。たとえば、プログラムの変更によって、インスタンス化が不要になり、記憶領域が無駄になることがあります。CCadmin -clean コマンド (以前のリリースの ptclean) を使用すれば、すべてのインスタンス化と関連データを整理できます。インスタンス化は、必要なときだけ再作成されます。

生成されるインスタンス

コンパイラは、テンプレートインスタンス生成のため、インラインテンプレート関数をインライン関数として扱います。コンパイラは、インラインテンプレート関数を他のインライン関数と同じように管理します。この章の内容は、テンプレートインライン関数には適用されません。

全クラスインスタンス化

コンパイラは通常、テンプレートクラスのメンバーを他のメンバーからは独立してインスタンス化するので、プログラム内で使用されるメンバーだけがインスタンス化されます。デバッガによる使用を目的としたメソッドは、通常はインスタンス化されません。

デバッグ中のメンバーを、デバッガから確実に利用できるようにするということは、次の 2 つを行うことになります。

- 第 1 に、実際には使用されないテンプレートクラスインスタンスメンバーを使用する、非テンプレート関数を作成します。この関数は呼び出されないようにする必要があります。
- 第 2 に、-template=wholeclass コンパイラオプションを使用します。このオプションを指定すると、非テンプレートで非インラインのメンバーのうちのどれかがインスタンス化された場合に、他の非テンプレート、非インラインのメンバーもすべてインスタンス化されます。

ISO C++ 標準では、特定のテンプレート引数により、すべてのメンバーが正当であるとはかぎらないテンプレートクラスを作成してよいと規定しています。不正メンバーをインスタンス化しないかぎり、プログラムは依然として適正です。ISO C++ 標準ライブラリでは、この技法が使用されています。ただし、-template=wholeclass オ

プションはすべてのメンバーをインスタンス化するので、問題のあるテンプレート引数を使ってインスタンス化する場合には、この種のテンプレートクラスに使用できません。

コンパイル時のインスタンス化

インスタンス化とは、C++ コンパイラがテンプレートから使用可能な関数やオブジェクトを作成するプロセスをいいます。C++ コンパイラ ではコンパイル時にインスタンス化を行います。つまり、テンプレートへの参照がコンパイルされているときに、インスタンス化が行われます。

コンパイル時のインスタンス化の長所を次に示します。

- デバッグが非常に簡単である。エラーメッセージがコンテキストの中に発生するので、コンパイラが参照位置を完全に追跡することができる。
- テンプレートのインスタンス化が常に最新である。
- リンク段階を含めて全コンパイル時間が短縮される。

ソースファイルが異なるディレクトリに存在する場合、またはテンプレートシンボルを指定してライブラリを使用した場合には、テンプレートが複数回にわたってインスタンス化されることがあります。

テンプレートインスタンスの配置とリンケージ

バージョン 5.5 以降の Sun の C++ コンパイラの場合、インスタンスは特別なアドレスセクションに入り、重複しているものをリンカーが見つけて破棄します。コンパイラには、インスタンスの配置とリンケージの方法として、外部、静的、大域、明示的、半明示的のどれを使うかを指定できます。

- 外部インスタンスは、大半のプログラム開発に適していますが、以下の条件を満たす状況に最適です。
  - プログラムに含まれているインスタンス全体は小さいが、各コンパイル単位がそれぞれ参照するインスタンスが大きい。
  - 2、3 個以上のコンパイル単位で参照されるインスタンスがほとんどない。
- 静的、非推奨 - 下記参照

- デフォルトである大域インスタンスは、あらゆる開発に適していますが、さまざまなインスタンスをオブジェクトが参照する場合に最適です。
- 明示的インスタンスは、厳密に管理されたアプリケーションコンパイル環境に適しています。
- 半明示的インスタンスは、上記より多少管理の程度が緩やかなアプリケーションコンパイル環境に適しています。ただし、このインスタンスは明示的インスタンスより大きなオブジェクトファイルを生成し、用途は限られています。

この節では、5 つのインスタンスの配置とリンケージの方法について説明します。インスタンスの生成に関する詳細は、70 ページの「テンプレートのインスタンス化」にあります。

外部インスタンス

外部インスタンスの場合では、すべてのインスタンスがテンプレートリポジトリ内に置かれます。テンプレートインスタンスは 1 つしか存在できません。つまり、インスタンスが未定義であるとか、重複して定義されているということはありません。テンプレートは必要な場合にのみ再インスタンス化されます。非デバッグコードの場合、すべてのオブジェクトファイル (テンプレートキャッシュに入っているものを含む) の総サイズは、`-instances=extern` を指定したときの値が `-instances=global` を指定したときの値より小さくなることがあります。

テンプレートインスタンスは、リポジトリ内では大域リンケージを受け取ります。インスタンスは、現在のコンパイル単位からは、外部リンケージで参照されます。

注 – コンパイルとリンクを別々に行うとき、コンパイル処理で `-instance=extern` を指定した場合には、リンク処理でも`-instance=extern` を指定する必要があります。

この方法にはキャッシュが壊れる恐れがあるという欠点があります。そのため、別のプログラムに替えたり、大幅な変更をプログラムに対して行なったりした場合にはキャッシュをクリアーする必要があります。キャッシュへのアクセスを一度に 1 回だけに限定しなければならないため、キャッシュは、`dmake` を使用する場合と同じように、並列コンパイルにおけるボトルネックとなります。また、1 つのディレクトリ内に構築できるプログラムは 1 個だけです。

メインオブジェクトファイル内にインスタンスを作成した後必要に応じて破棄するよりも、有効なテンプレートインスタンスがすでにキャッシュに存在しているかどうかを確認するほうが、時間がかかる可能性があります。

外部リンケージは、-instances=extern オプションによって指定します。

インスタンスはテンプレートリポジトリ内に保存されているので、外部インスタンスを使用する C++ オブジェクトをプログラムにリンクするには CC コマンドを使用しなければなりません。

使用するすべてのテンプレートインスタンスを含むライブラリを作成したい場合には、-CC コマンドに -xar オプションを指定してください。ar コマンドは使用できません。例を次に示します。

```
example% CC -xar -instances=extern -o libmain.a a.o b.o c.o
```

詳細については、第 16 章を参照してください。

キャッシュの衝突

-instance=extern を指定する場合、キャッシュの衝突の可能性があるため、異なるバージョンのコンパイラを同一ディレクトリ内で実行しないでください。
-instances=extern テンプレートモデルを使用する場合は、以下の点に注意してください。

- 同一ディレクトリ内に、無関係のバイナリを作成しないでください。同一ディレクトリ内に作成されたバイナリ (.o、.a、.so、実行可能プログラム) はすべて関連している必要があります。これは、複数のオブジェクトファイルに共通のすべてのオブジェクト、関数、型の名前は、定義が同一であるためです。
- dmake を使用する場合などは、複数のコンパイルを同一ディレクトリで同時に実行しても問題はありません。他のリンク段階と同時にコンパイルまたはリンク段階を実行すると、問題が発生する場合があります。リンク段階とは、ライブラリまたは実行可能プログラムを作成する処理を意味します。makefile 内での依存により、1 つのリンク段階での並列実行が禁止されていることを確認してください。

静的インスタンス

注 – `-instances=static` オプションは、非推奨です。`-instances=global` が `static` の利点をすべて備えており、かつ欠点を備えていないので、`-instances=static` を使用する理由はなくなっています。このオプションは、C++ 5.5 には存在しない、旧リリースのコンパイラにあった問題を克服するために用意されていました。

静的インスタンスの場合は、すべてのインスタンスが現在のコンパイル単位内に置かれます。その結果、テンプレートは各再コンパイル作業中に再インスタンス化されます。インスタンスはテンプレートリポジトリに保存されません。

この方法の利点は、コンパイル単位から呼び出されたインスタンスはそのコンパイル単位によってコンパイルされること、したがってそれらのインスタンスがデバッグ可能であることを確保することによって、デバッグ処理に役立つ点です。`-instances=global` を使用した場合、この点は保証されません。

この方法の欠点は、言語の意味解釈が規定どおりでないこと、かなり大きいオブジェクトと実行可能ファイルが作られることです。

インスタンスは静的リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位以外では認識することも使用することもできません。そのため、テンプレートの同じインスタンス化がいくつかのオブジェクトファイルに存在することがあります。複数のインスタンスによって不必要に大きなプログラムが生成されるので、静的インスタンスのリンケージは、テンプレートがインスタンス化される回数が少ない小さなプログラムだけに適しています。

静的インスタンスは潜在的にコンパイル速度が速いため、修正継続機能を使用したデバッグにも適しています (『dbx コマンドによるデバッグ』を参照してください)。

注 – プログラムがコンパイル単位間で (テンプレートクラスまたはテンプレート機能の静的データメンバー) テンプレートインスタンスの共有に依存している場合は、静的インスタンス方式は使用しないでください。プログラムが正しく動作しなくなります。

静的インスタンスリンケージは、`-instances=static` コンパイルオプションで指定します。

大域インスタンス

旧リリースのコンパイラとは異なり、新リリースでは、大域インスタンスの複数のコピーを防ぐ必要はなくなっています。

この方法の利点は、他のコンパイラで通常受け入れられる正しくないソースコードを、このモードで受け入れられるようになったという点です。特に、テンプレートインスタンスの中からの静的変数への参照は正当なものではありませんが、通常は受け入れられるものです。

この方法の欠点は、テンプレートインスタンスが複数のファイルにコピーされることから、個々のオブジェクトファイルが通常より大きくなる可能性がある点です。デバッグを目的としてオブジェクトファイルの一部を -g オプションを使ってコンパイルし、他のオブジェクトファイルを-g オプションなしでコンパイルした場合、プログラムにリンクされるテンプレートインスタンスが、デバッグバージョンと非デバッグバージョンのどちらであるかを予測することは難しくなります。

テンプレートインスタンスは大域リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位の外でも認識でき、使用できます。

大域インスタンスは、`-instances=global` オプションで指定します (これがデフォルトです)。

明示的インスタンス

明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されたテンプレートに対してのみ生成されます。暗黙的なインスタンス化は行われません。インスタンスは現在のコンパイル単位に置かれます。したがって、テンプレートは再コンパイルごとに再インスタンス化され、テンプレートリポジトリには保存されません。

この方法の利点は、生成される明示的インスタンスが 1 つだけであることを保証する必要がない点です。テンプレートのコンパイル量もオブジェクトのサイズも、他のどの方法より小さくて済みます。

欠点は、すべてのインスタンス化を手動で行わなければならないという点です。

テンプレートインスタンスは大域リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位の外でも認識でき、使用できます。リンカーは、重複しているものを見つけ、破棄します。

明示的インスタンスは、`.-instances=explicit` オプションで指定します。

半明示的インスタンス

半明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されるテンプレートやテンプレート本体の中で暗黙的にインスタンス化されるテンプレートに対してのみ生成されます。明示的に作成されるインスタンスで必要となるインスタンスは自動的に生成されますが、内部 (静的) リンケージが与えられます。main コード行内で行う暗黙的なインスタンス化は不完全になります。インスタンスは現在のコンパイル単位に置かれます。したがって、テンプレートは再コンパイルごとに再インスタンス化され、テンプレートリポジトリには保存されません。

インスタンスは大域リンケージを受け取ります。これらのインスタンスは、現在のコンパイル単位の外でも認識でき、使用できます。リンカーは、重複しているものを見つけ、破棄します。

半明示的インスタンスは、-instances=semiexplicit オプションで指定します。

テンプレートリポジトリ

必要なときだけテンプレートインスタンスがコンパイルされるよう、コンパイルからコンパイルまでのテンプレートインスタンスが テンプレートリポジトリに保存されます。テンプレートリポジトリには、外部インスタンスメソッドを使用するときにテンプレートのインスタンス化に必要となる非ソースファイルがすべて入っています。このリポジトリが他の種類のインスタンスに使用されることはありません。

リポジトリの構造

テンプレートリポジトリは、デフォルトで、キャッシュディレクトリ (SunWS_cache) にあります。

キャッシュディレクトリは、オブジェクトファイルが置かれるのと同じディレクトリ内にあります。SUNWS_CACHE_NAME環境変数を設定すれば、キャッシュディレクトリ名を変更できます。SUNWS_CACHE_NAME 変数の値は必ずディレクトリ名にし、パス名にしてはならない点に注意してください。これは、コンパイラが、テンプレートキャッシュディレクトリをオブジェクトファイルディレクトリの下に自動的に入れることから、コンパイラがすでにパスを持っているためです。

テンプレートリポジトリへの書き込み

コンパイラは、テンプレートインスタンスを格納しなければならないとき、出力ファイルに対応するテンプレートリポジトリにそれらを保存します。たとえば、以下のコマンド行は、./sub/a.o というオブジェクトファイルを生成し、./sub/SunWS_cashe ディレクトリ内のリポジトリにテンプレートインスタンスを書き込みます。 コンパイラがテンプレートをインスタンス化するときにこのキャッシュディレクトリが存在しない場合は、このディレクトリが作成されます。

```
example% CC -o sub/a.o a.cc
```

複数のテンプレートリポジトリからの読み取り

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。たとえば次の例では、./sub1/SunWS_cache と ./sub2/SunWS_cache から読み取り、必要に応じて ./SunWS_cache へ書き込みます。

```
example% CC sub1/a.o sub2/b.o
```

テンプレートリポジトリの共有

リポジトリ内にあるテンプレートは、ISO/ANSI C++ 標準の単一定義規則に違反してはなりません。つまり、テンプレートは、どの用途に使用される場合でも、1 つのソースから派生したものでなければなりません。この規則に違反した場合の動作は定義されていません。

この規則に違反しないようにするための (もっとも保守的で) もっとも簡単な方法は、1 つのディレクトリ内では 1 つのプログラムまたはライブラリしか作成しないことです。無関係な 2 つのプログラムが同じ型名または外部名を使用して別のものを意味する場合があります。これらのプログラムがテンプレートリポジトリを共有すると、テンプレートの定義が競合し、予期せぬ結果が生じる可能性があります。

-instance=extern による テンプレートインスタンスの自動一貫性

-instances=extern を指定すると、テンプレートリポジトリマネージャは、リポジトリ中のインスタンスの状態をソースファイルと確実に一致させて最新の状態にします。

たとえば、ソースファイルが-g オプション (デバッグ付き) でコンパイルされる場合には、データベースの中の必要なファイルも-g でコンパイルされます。

さらに、テンプレートリポジトリはコンパイル時の変更を追跡します。たとえば、-DDEBUG フラグを指定して名前DEBUG を定義すると、データベースがこれを追跡します。その次のコンパイルでこのフラグを省くと、コンパイラはこの依存性が設定されているテンプレートを再度インスタンス化します。

テンプレート定義の検索

定義分離型のテンプレートの編成 (テンプレートを使用するファイルの中にテンプレートの宣言だけがあって定義はないという編成) を使用している場合には、現在のコンパイル単位にテンプレート定義が存在しないので、コンパイラが定義を検索しなければなりません。この節では、そうした検索について説明します。

定義の検索はかなり複雑で、エラーを発生しやすい傾向があります。このため、可能であれば、定義取り込み型のテンプレートファイルの編成を使用したほうがよいでしょう。こうすれば、定義検索をまったく行わなくて済みます。62 ページの「テンプレート定義の取り込み」を参照してください。

注 – -template=no%extdef オプションを使用する場合、コンパイラは別のソースファイルを検索しません。

ソースファイルの位置規約

オプションファイルで提供されるような特定の指令がない場合には、コンパイラは Cfront 形式の方法でテンプレート定義ファイルを検出します。この方法の場合、テンプレート宣言ファイルと同じベース名がテンプレート定義ファイルに含まれている必要があります。また、テンプレート定義ファイルが現在の include パス上に存在

している必要もあります。たとえば、テンプレート関数 foo() が foo.h 内にある場合には、それと一致するテンプレート定義ファイルの名前を foo.cc か、または他の何らかの認識可能なソースファイル拡張子 (.C、.c、.cc、.cpp、.cxx、または .c++) にしなければなりません。テンプレート定義ファイルは、通常使用する include ディレクトリの 1 つか、またはそれと一致するヘッダーファイルと同じディレクトリの中に置かなければなりません。

定義検索パス

-I で設定する通常の検索パスの代わりに、-pti*directory* オプションでテンプレート定義ファイルの検索ディレクトリを指定することができます。複数の -pti フラグは、複数の検索ディレクトリ、つまり 1 つの検索パスを定義します。-pti*directory* を使用している場合には、コンパイラはこのパス上のテンプレート定義ファイルを探し、-I フラグを無視します。しかし、-pti*directory* フラグはソースファイルの検索規則を複雑にするので、-pti*directory* フラグより -I フラグを使用してください。

テンプレートオプションファイル

テンプレートオプションファイルとは、テンプレート定義を特定したり、インスタンスを再コンパイルする際に必要なオプションを含む、ユーザーが用意するファイルです (省略も可)。このファイルを使ってテンプレートの特殊化と明示的なインスタンス化を制御することもできます。しかし、現在特殊化の宣言と明示的なインスタンス化に必要な構文はソースコード中で使用できるため、テンプレートオプションをこの用途に使用すべきではありません。

注 – テンプレートオプションファイルは、C++ コンパイラの将来のリリースではサポートされなくなります。

オプションファイルの名前は CC_tmpl_opt で、SunWS_config ディレクトリにあります。オプションファイルは ASCII テキストファイルで、多くのエントリを含んでいます。エントリはキーワードから始まり、テキストが続き、セミコロン (;) で終わります。エントリは複数行に渡ってもかまいませんが、キーワードは必ず 1 行中に収めるようにしてください。分割してはなりません。

注釈

コメントは # 文字で始まり、その行の終わりまで続きます。コメント内のテキストは無視されます。

```
# コメントのテキストは行末まで無視されます。
```

インクルード

オプションファイルをインクルードすれば、複数のテンプレートデータベース間でオプションファイルを共有できます。この機能は特に、テンプレートを含むライブラリを構築するときに便利です。処理中、指定されたオプションファイルは原文どおりに現在のオプションファイルにインクルードされます。オプションファイル内では、複数の include 文をどこにでも指定できます。オプションファイルは入れ子にすることもできます。

```
include "options-file";
```

ソースファイルの拡張子

コンパイラがデフォルトの Cfront 形式のソースファイル検索機構を使用している場合、extensions エントリを使用すれば、コンパイラが検索するソースファイルの拡張子を指定できます。次に、このエントリの構文を示します。

```
extensions "ext-list";
```

ext-list には有効なソースファイルの拡張子を、空白文字で区切って指定します。

```
extensions ".CC .c .cc .cpp";
```

このエントリがオプションファイルに存在しない場合、コンパイラが検索する拡張子は、.cc、.c、.cpp、.C 、.cxx および .c++ です。

定義ソースの位置

定義ソースファイルの位置は、オプションファイル中のdefinition エントリで明示的に指定できます。definition エントリは、テンプレートの宣言と定義のファイル名が標準の Cfront 形式の規約に準拠していない場合に使用してください。次に、このエントリの構文を示します。

```
definition name in "file-1",[ "file-2" ..., "file-n"] [nocheck "options"];
```

name フィールドには、このエントリでの指定を適用するテンプレートを指定します。1 つの *name* に使用できる definition エントリは 1 つだけです。*name* に指定する名前は単純な名前である必要があります。つまり、修飾名は使用できません。また、丸かっこ、戻り型、およびパラメータリストも使用できません。戻り型やパラメータに関わらず、名前そのものだけが重要です。結果として、definition エントリは複数の (おそらくは、多重定義された) テンプレートに適用される可能性があります 。

「*file-n*」リストフィールドには、テンプレート定義が含まれているファイルを指定します。ファイルの検索には、定義検索パスが使用されます。ファイル名は引用符 (") で囲む必要があります。複数のファイルを指定する理由は、指定した単純なテンプレート名がファイルごとに定義されている複数の異なるテンプレート名を参照していたり、1 つのテンプレートの定義がファイルごとに異なっている可能性があるためです。たとえば、func が 3 つのファイルで定義されている場合、これら 3 つのファイルを definition エントリのリストに指定する必要があります。

nocheck フィールドについては、この節の最後で説明します。

次の例では、コンパイラは foo.cc にあるテンプレート関数 foo を見つけ、この関数をインスタンス化します。ただし foo.cc 中の関数 foo はデフォルトの検索でも検出されるため、この definition エントリは冗長です。

コード例 7-1　冗長な definition エントリ

```
foo.cc                  template <class T> T foo( T t ) { }
CC_tmpl_opt             definition foo in "foo.cc";
```

次の例では、静的なデータメンバーの定義と単純名の使用を示します。

コード例 7-2　静的なデータメンバーの定義と単純名の使用

```
foo.h                    template <class T> class foo { static T* fooref; };

foo_statics.cc           #include "foo.h"
                         template <class T> T* foo<T>::fooref = 0

CC_tmpl_opt              definition fooref in "foo_statics.cc";
```

fooref の定義で使用されている名前は単純名であり、修飾名 (foo::fooref など) ではありません。この definition エントリを定義する理由は、ファイル名が認識可能な拡張子ではなく (foo.cc など)、デフォルトの Cfront 形式の検索規則ではファイルを見つけることができないためです。

次の例では、テンプレートメンバー関数の定義を示します。例に示すとおり、メンバー関数は静的メンバー初期設定子とまったく同じように処理されます。

コード例 7-3　テンプレートメンバー関数の定義

```
foo.h                    template <class T> class foo { T* foofunc(T); };

foo_funcs.cc             #include "foo.h"
                         template <class T> T* foo<T>::foofunc(T t) {}

CC_tmpl_opt              definition foofunc in "foo_funcs.cc";
```

次の例では、2 つの異なるソースファイルにあるテンプレート関数の定義を示します。

コード例 7-4　異なるソースファイルにあるテンプレート関数の定義

```
foo.h                    template <class T> class foo {
                             T* func( T t );
                             T* func( T t, T x );
                         };

foo1.cc                  #include "foo.h"
                         template <class T> T* foo<T>::func( T t ) { }

foo2.cc                  #include "foo.h"
                         template <class T> T* foo<T>::func( T t, T x ) { }

CC_tmpl_opt              definition func in "foo1.cc", "foo2.cc";
```

この例では、コンパイラは多重定義されている関数 func() の定義を両方とも見つける必要があります。そこで、definition エントリで、どこに適切な関数定義があるのかをコンパイラに指示します。

コンパイルフラグが変更されても、再コンパイルが不必要な場合もあります。オプションファイルのdefinition エントリに nocheck フィールドを指定すると、不必要な再コンパイルを回避できます。nocheck フィールドでオプションを指定すると、コンパイラとテンプレートデータベースマネージャは、そのオプションを依存関係の検査対象から除外します。特定のコマンド行フラグを追加または削除したために、コンパイラがテンプレート関数を再インスタンス化する必要がない場合は、nocheck フラグを使用してください。次に、このエントリの構文を示します。

```
definition name in "file-1"[, "file-2" ..., "file-n"] [nocheck "options"];
```

オプション (*options*) は引用符 (") で囲む必要があります。

次の例では、コンパイラは foo.cc にあるテンプレート関数 foo を見つけ、この関数をインスタンス化します。後で再インスタンス化のための検査が必要な場合、コンパイラは -g オプションを無視します。

コード例 7-5 nocheck オプション

```
foo.cc                  template <class T> T foo( T t ) {}
CC_tmpl_opt             definition foo in "foo.cc" nocheck "-g";
```

テンプレートの特殊化エントリ

最近まで、C++ 言語はテンプレートを特殊化するための機構を持っておらず、個々のコンパイラが独自の機能を提供していました。この節では、以前のバージョンの C++ コンパイラの機構を使用したテンプレートの特殊化を説明します。この機構は、互換モード (-compat[=4]) でのみサポートされています。

special エントリは、指定された関数が特殊なものであり、この関数に遭遇してもインスタンス化してはならないことをコンパイラに指示します。コンパイル時にインスタンス化する方法を使用する場合は、オプションファイルの中で special エントリを使用して、特殊化を事前に登録してください。次に、このエントリの構文を示します。

```
special declaration;
```

宣言（*declaration*）には、戻り型がない正しい C++ 形式の宣言を指定します。例を次に示します。

コード例 7-6 special エントリ

foo.h	`template <class T> T foo( T t ) { };`
main.cc	`#include "foo.h"`
CC_tmpl_opt	`special foo(int);`

上記の special エントリを含むオプションファイルは、テンプレート関数 foo() を int 型にインスタンス化してはならないこと、および、特殊化された関数 foo() がユーザーから提供されることをコンパイラに指示します。このエントリをオプションファイルに指定しない場合、関数は不必要に再インスタンス化され、その結果、エラーが発生します。

コード例 7-7 special エントリを使用する必要がある場合

foo.h	`template <classT> T foo( T t ) { return t + t; }`
file.cc	`#include "foo.h"` `int func( ) { return foo( 10 ); }`

main.cc	`#include "foo.h"`
	`int foo( int i ) { return i * i; } // 特殊化`
	`int main( ) { int x = foo( 10 ); int y = func();`
	`return 0; }`

上記の例では、main.cc をコンパイルするとき、コンパイラはその定義をあらかじめ確認しているため、特殊化された foo を正しく使用します。しかし、file.cc をコンパイルするとき、コンパイラは main.cc に foo が存在することを知らないため、foo に対して独自のインスタンス化を行います。この結果、ほとんどの場合は、この

リンク中にシンボルが複数回定義されるだけですが、場合によっては (特にライブラリの場合)、間違った関数が使用され、実行時エラーが発生することがあります。特殊化された関数を使用する場合は、その特殊化を登録しておくことをお勧めします。

special エントリは多重定義できます。次に例を示します。

コード例 7-8 special エントリの多重定義

foo.h	`template <classT> T foo( T t ) {}`

main.cc	`#include "foo.h"`
	`int foo( int i ) {}`
	`char* foo( char* p ) {}`

CC_tmpl_opt	`special foo(int);` `special foo(char*);`

テンプレートクラスを特殊化するには、special エントリにテンプレート引数を指定します。

コード例 7-9 テンプレートクラスの特殊化

foo.h	`template <class T> class Foo { ... various members ... };`
main.cc	`#include "foo.h"` `int main( ) { Foo<int> bar; return 0; }`
CC_tmpl_opt	`special class Foo<int>;`

テンプレートクラスメンバーが静的なメンバーの場合、special エントリにキーワード static を指定する必要があります。

コード例 7-10 静的テンプレートクラスメンバーの特殊化

foo.h	`template <class T> class Foo { public: static T func(T);` `};`
main.cc	`#include "foo.h"` `int main( ) { Foo<int> bar; return 0; }`
CC_tmpl_opt	`special static Foo<int>::func(int);`

第8章

例外処理

この章では、C++ コンパイラの例外処理の実装について説明します。129 ページの「マルチスレッドプログラムでの例外の使用」にも補足情報を掲載しています。例外処理の詳細については、『プログラミング言語 C++ 』(第 3 版 Bjarne Stroustrup 著、アスキー、1997 年) を参照してください。

同期例外と非同期例外

例外処理では、配列範囲のチェックといった同期例外だけがサポートされます。同期例外とは、例外を `throw` 文からだけ生成できることを意味します。

C++ 標準でサポートされる同期例外処理は、終了モデルに基づいています。終了とは、いったん例外が送出されると、例外の送出元に制御が二度と戻らないことを意味します。

例外処理では、キーボード割り込みなどの非同期例外の直接処理は行えません。ただし、注意して使用すれば、非同期イベントが発生したときに、例外処理を行わせることができます。たとえば、シグナルに対する例外処理を行うには、大域変数を設定するシグナルハンドラと、この変数の値を定期的にチェックし、値が変化したときに例外を送出するルーチンを作成します。シグナルハンドラからは例外を送出できません。

実行時エラーの指定

例外に関する実行時エラーメッセージには、次の 5 種類があります。

- 例外のハンドラがありません
- 予期しない例外を送出
- ハンドラでは例外の再送出しかできません
- スタックの巻き戻し中は、デストラクタは独自の例外を処理しなければなりません
- メモリー不足

実行時にエラーが検出されると、現在の例外の種類と、上の 5 つのメッセージのいずれかがエラーメッセージとして表示されます。デフォルト設定では、事前定義済みの `terminate()` 関数が呼び出され、さらにこの関数から `abort()` が呼び出されます。

コンパイラは、例外指定に含まれている情報に基づいて、コードの生成を最適化します。たとえば、例外を送出しない関数のテーブルエントリは抑止されます。また、関数の例外指定の実行時チェックは、できるかぎり省略されます。

例外の無効化

プログラムで例外を使用しないことが明らかであれば、`-features=no%except` コンパイラオプションを使用して、例外処理用のコードの生成を抑止することができます。このオプションを使用すると、コードサイズが若干小さくなり、実行速度が多少高速になります。ただし、例外を無効にしてコンパイルしたファイルを、例外を使用するファイルにリンクすると、例外を無効にしてコンパイルしたファイルに含まれている局所オブジェクトが、例外が発生したときに破棄されずに残ってしまう可能性があります。デフォルト設定では、コンパイラは例外処理用のコードを生成します。時間と容量のオーバーヘッドが重要な場合を除いて、通常は例外を有効のままにしておいてください。

注 – C++ 標準ライブラリ、`dynamic_cast`、デフォルトの `new` 演算子では例外が必要です。そのため、標準モード (デフォルトモード) でコンパイルを行う場合は、例外を無効にしないでください。

実行時関数と事前定義済み例外の使用

標準ヘッダー <exception> には、C++ 標準で示されている各種のクラスと例外用の関数が含まれています。このヘッダーは、標準モード (コンパイラのデフォルトモード、すなわち -compat=5 オプションを使用するモード) でコンパイルを行うときだけ使用されます。以下は、<exception> ヘッダーファイルの宣言を抜粋したものです。

```
// 標準ヘッダー <exception>
namespace std {
    class exception {
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();    };
    class bad_exception: public exception { ... };
    // 予期しない例外処理
    typedef void (*unexpected_handler)();
    unexpected_handler
     set_unexpected(unexpected_handler) throw();
    void unexpected();
    // 終了処理
    typedef void (*terminate_handler)();
    terminate_handler set_terminate(terminate_handler) throw();
    void terminate();
    bool uncaught_exception() throw();
}
```

標準クラス exception は、構文要素や C++ 標準ライブラリから送出されるすべての例外のための基底クラスです。exception 型のオブジェクトは、例外を発生させることなく作成、複製、破棄することができます。仮想メンバー関数 what() は、例外についての情報を示す文字列を返します。

C++ リリース 4.2 で使用される例外との互換性を保つため、標準モードで使用する <exception.h> というヘッダーも用意されています。このヘッダーは、C++ 標準のコードに移行するためのもので、C++ 標準には含まれていない宣言を含んでいます。開発スケジュールに余裕があれば、(<exception.h> の代わりに <exception> を使用し) コードを C++ 標準に従って書き換えてください。

```
// ヘッダー <exception.h>、移行用
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

互換モード (:compat[=4]) では、ヘッダー <exception> は使用できません。また、ヘッダー <exception.h> は、C++ リリース 4.2 で提供されているヘッダーと同じですので、ここでは取り上げません。

シグナルや Setjmp/Longjmp と例外との併用

同じプログラムの中で、setjmp/longjmp 関数と例外処理を併用することができます。ただし、これらが相互に干渉しないことが条件になります。

その場合、例外と setjmp/longjmp のすべての使用規則が、それぞれ別々に適用されます。また、A 地点から B 地点への longjmp を使用できるのは、例外を A 地点から送出し、B 地点で捕獲した場合と効果が同じになる場合だけです。特に、try ブロックへの、または try ブロックからの (直接的または間接的な) longjmp や、自動変数や一時変数の初期化や明示的な破棄の前後にまたがる longjmp は行なってはいけません。

シグナルハンドラからは例外を送出できません。

例外のある共有ライブラリの構築

C++ コードが含まれているプログラムでは、-Bsymbolic を使用せずに、リンカーのマップファイルを使用してください。-Bsymbolic を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

dlopen を使用して共有ライブラリを開いた場合は、RTLD_GLOBAL を使用しないと例外が機能しません。

第9章

キャスト演算

この章では、C++ 標準の新しいキャスト演算子、すなわち const_cast、reinterpret_cast、static_cast、dynamic_cast について説明します。キャストとは、オブジェクトや値の型を、別の型に変換することです。

これらのキャスト演算子を使用すると、従来のキャスト演算子よりも緻密な制御を行うことができます。たとえば、dynamic_cast<> 演算子では、多相クラスのポインタの実際の型をチェックすることができます。新形式のキャストには、(_cast を検索することで) テキストエディタで簡単に検出できるという利点もあります。従来のキャストは、構文チェックを行わないと検出できません。

新しいキャストは、それぞれ従来のキャスト表記で行うことのできる各種のキャスト操作の一部だけを実行します。たとえば、const_cast<int*>(v) は、従来であれば (int*)v と記述することができます。新しいキャストは、コードの意図をより明確に表現し、コンパイラがより的確なチェックを行えるように、実行可能な各種のキャスト操作を単に類別したものです。

キャスト演算子は常に有効になります。これらを無効にすることはできません。

cast キャスト

式const_cast<*T*>(*v*) を使用して、ポインタまたは参照の const 修飾子または volatile 修飾子を変更することができます (新しい形式のキャストのうち、const 修飾子を削除できるのは const_cast<> のみ)。

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_castconst_cast<A&>(a1).f( );              // const を削除
ip = const_cast<int*> (cvip);          // const と volatile を削除
}
```

解釈を変更するキャスト

式 reinterpret_cast<*T*>(*v*) は式 *v* の値の解釈を変更します。この式は、ポインタ型と整数型の間、2 つの無関係なポインタ型の間、ポインタ型からメンバー型へ、ポインタ型から関数型へ、という各種の変換に使用できます。

reinterpret_cast 演算子を使用すると、未定義の結果または実装に依存しない結果を出すことがあります。次に、確実な動作について説明します。

- データオブジェクトまたは関数へのポインタ (メンバーへのポインタは除く) は、それを十分保持できる大きさの任意の整数型に変換できます (long 型は十分大きいため、C++ がサポートするアーキテクチャでは常にポインタ値を保持できます)。
- (非メンバー) 関数へのポインタは、別の (非メンバー) 関数型へのポインタに変換できます。元の型に戻しても、値は元の値と同じになります。

- 新しい型が元の型よりも厳しい整列条件を持たない場合、オブジェクトへのポインタは別のオブジェクト型へのポインタに変換できます。元の型に戻しても、値は元の値と同じになります。
- reinterpret_cast 演算子を使用して型「*T1* のポインタ」の式を型「*T2* のポインタ」に変換できる場合、型 *T1* の左辺値は型「*T2* の参照」に変換できます。
- *T1* と *T2* の両方が関数型であるか両方がオブジェクト型である場合、「型 *T1* の X のメンバーを指すポインタ」型の右辺値は、「型 *T2* の Y のメンバーを指すポインタ」型の右辺値に明示的に変換できます。
- (変換が許可されている場合) ある型のヌルポインタは別の型のヌルポインタに変換された後もヌルポインタのままです。
- reinterpret_cast 演算子を使用して、const を const でない型にキャストすることはできません。このようにキャストするには const キャストを使用します。
- reinterpret_cast 演算子を使用して、ポインタと、同じクラス階層に存在する別のクラスの間の変換を行うことはできません。このように変換するには、静的キャストまたは動的キャストを使用します (reinterpret_cast は必要があっても調整は行わない)。

```
class A { int a; public: A(); };
class B : public A { int b, c; };
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);       // 安全
     B* bp = reinterpret_cast<B*>(&a1);     // 安全ではない
     const A a2;
     ap = reinterpret_cast<A*>(&a2);        // エラー、const が削除
された
}
```

静的キャスト

式 static_cast<*T*>(*v*) は、式の値 *v* を型 *T* の値に変換します。この式は、暗黙的に実行されるすべての型変換に使用できます。さらに、いかなる値でもvoid にキャストすることができ、いかなる暗黙的型変換でも、そのキャストが旧式のキャストと同様に正当である限り、反転させることができます。

```
class B             { ... };
class C : public B { ... };
enum E { first=1, second=2, third=3 };
void use_of_static_cast(C* c1 )
{
  B* bp = c1;                    // 暗黙的な変換
  C* c2 = static_cast<C*>(bp); // 暗黙的な変換を反転させる
  int i = second;                // 暗黙的な変換
  E e = static_cast<E>(i);     // 暗黙的な変換を反転させる
}
```

static_cast 演算子を使用して、const を const 以外の型にするようなキャストを行うことはできません。階層の下位に (基底から派生ポインタまたは参照へ) キャストするには static_cast を使用できますが、変換は検証されず、結果は使用できない場合があります。抽象基底クラスから下位へのキャストには、static_cast は使用できません。

動的キャスト

クラスへのポインタ (または参照) は、そのクラスから派生されたすべてのクラスを実際に指す (参照する) ことができます。場合によっては、オブジェクトの完全派生クラス、またはその完全なオブジェクトの他のサブオブジェクトへのポインタを得る方が望ましいことがあります。動的キャストによってこれが可能になります。

注 – 互換モード (-compat[=4]) でコンパイルする場合、プログラムが動的キャストを使用している場合は、-features=rtti を付けてコンパイルする必要があります。

動的な型のキャストは、あるクラス *T1* へのポインタ (または参照) を別のクラス *T2* のポインタ (または参照) に変換します。*T1* と *T2* は、同じ階層内になければなりません。両クラスとも (公開派生を介して) アクセス可能でなければならず、変換はあいまいであってはなりません。また、変換が派生クラスからその基底クラスの 1 つに対するものでないかぎり、*T1* と *T2* の両方が入った階層の最小の部分は多相性がなければなりません (少なくとも仮想関数が 1 つ存在すること)。

式`dynamic_cast<`*T*`>(`*v*`)` では、*v* はキャストされる式であり、*T* はキャストの対象となる型です。*T* は完全なクラス型 (定義が参照できるもの) へのポインタまたは参照であるか、あるいは「*cv* `void` へのポインタ」でなければなりません。ここで*cv* は空の文字列、`const`、`volatile`、`const volatile` のいずれかです。

階層の上位にキャストする

階層の上位にキャストする場合で、*v* が指す (参照する) 型の基底クラスを *T* が指す (あるいは参照する) 場合、変換は `static_cast<`*T*`>(`*v*`)` で行われるものと同じです。

`void*` にキャストする

T が `void*` の場合、結果はオブジェクト全体のポインタになります。つまり、*v* はあるオブジェクト全体の基底クラスの 1 つを指す可能性があります。この場合、`dynamic_cast<void*>(`*v*`)` の結果は、*v* をオブジェクト全体の型 (種類は問わない) に変換した後で `void*` に変換した場合と同じです。

`void*` にキャストする場合、階層に多相性がなければなりません (仮想関数が存在すること)。結果は実行時に検証されます。

階層の下位または全体にキャストする

階層の下位または全体にキャストする場合、階層に多相性がなければなりません (仮想関数を持つ必要がある)。結果は実行時に検証されます。

階層の下位または全体にキャストする場合、*v* から *T* に変換できないことがあります。たとえば、試行された変換があいまいであったり、*T* に対するアクセスが不可能であったり、あるいは必要な型のオブジェクトを *v* が指さない (あるいは参照しない) 場合がこれに当たります。実行時検査が失敗し、*T* がポインタ型である場合、キャスト式の値は型 *T* のヌルポインタです。*T* が参照型の場合、何も返されず (C++ にはヌル参照は存在しない)、標準例外 `std::bad_cast` が送出されます。

たとえば、次の公開派生のコード例は正常に実行されます。

```
#include <assert.h>
#include #include <stddef.h> // NULL 用

class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, public B { };

void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = &ab;        // キャストは不要
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // 成功
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

これに対して、次のコード例は正しく実行されません。基底クラス B にアクセスできないからです。

```
#include <assert.h>
#include <stddef.h> // NULL 用
#include <typeinfo>

class A { public: virtual void f() { } };
class B { public: virtual void g() { } };
class AB : public virtual A, private B { };

void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab; // 中断を防ぐため、C 形式のキャストが必要
  A* ap  = dynamic_cast<A*>(bp); // 失敗、B にアクセスできない
  assert(ap == NULL);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // 失敗、B にアクセスできない
  }
  catch(const std::bad_cast&) {
    return; // 参照キャストの失敗をここで捕獲
  }
  assert(0); // ここまで到達しない
}
```

1 つの基底クラスについて仮想継承と多重継承が存在する場合には、実際の動的キャストは一意の照合を識別することができなければなりません。もし照合が一意でないならば、そのキャストは失敗します。たとえば、下記の追加クラス定義が与えられたとします。

```
class AB_B :     public AB,        public B  { };
class AB_B__AB : public AB_B,      public AB { };
```

例:

```
void complex_dynamic_casts( )
{
  AB_B__AB ab_b__ab;
  A*ap = &ab_b__ab;
                    // okay: finds unique A statically
  AB*abp = dynamic_cast<AB*>(ap);
                    // fails: ambiguous
  assert( abp == NULL );
                    // STATIC ERROR: AB_B* ab_bp = (AB_B*)ap;
                    // not a dynamic cast
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // dynamic one is okay
  assert( ab_bp != NULL );
}
```

dynamic_cast のエラー時のヌル (NULL) ポインタの戻り値は、コード中の 2 つのブロック (1 つは型推定が正しい場合にキャストを処理するためのもの、もう 1 つは正しくない場合のもの) の間の条件として役立ちます。

```
void using_dynamic_cast( A* ap )
{
  if ( AB *abp = dynamic_cast<AB*>(ap) )
    {    // abp は NULL ではない。
                // したがって ap は AB オブジェクトへのポインタである。
                 // abp を使用する。
      process_AB( abp ); }
  else
    {    // abp は NULL である。
              // したがって ap は AB オブジェクトへのポインタではない。
               // abp は使用しない。
      process_not_AB( ap );
    }
}
```

互換モード (-compat[=4]) では、-features=rtti コンパイラオプションによって実行時の型情報が有効になっていないと、コンパイラは dynamic_cast を static_cast に変換し、警告メッセージを出します。

実行時型情報が無効にされている場合、すなわち -features=no%rtti の場合には 、コンパイラは dynamic_cast<T&> を static_cast<T&> に変換し、警告を発行します。参照型への動的キャストを行う場合は、そのキャストが実行時に無効であると判明したときに送出される例外が必要です。例外に関する情報は、第 8 章を参照してください。

動的キャストは必然的に、仮想関数による変換のような適切な設計パターンより遅くなります。Erich Gamma 著 (ソフトバンク)『オブジェクト指向における再利用のためのデザインパターン』を参照してください。

第10章

プログラムパフォーマンスの改善

C++ 関数のパフォーマンスを高めるには、コンパイラが C++ 関数を最適化しやすいように関数を記述することが必要です。言語一般、特に C++ のソフトウェアパフォーマンスについて関連する書籍は多数あります。たとえば、『C++ Programming Style』(Tom Cargill 著、Addison-Wesley、1992 年発行)、『Writing Efficient Programs』(Jon Louis Bentley 著、Prentice-Hall、1982 年発行)、『Efficient C++: Performance Programming Techniques(Dov Bulka と David Mayhew 共著、Addison-Wesley、2000 年発行)、『Effective C++』(改訂 2 版、Scott Meyers 著、アスキー、1998 年) を参照してください。この章では、これらの書籍にある内容を繰り返すのではなく、Sun C++ コンパイラにとって特に有効なパフォーマンス向上の手法について説明します。

一時オブジェクトの回避

C++ 関数は、暗黙的に一時オブジェクトを多数生成することがよくあります。これらのオブジェクトは、生成後破棄する必要があります。しかし、そのようなクラスが多数ある場合は、この一時的なオブジェクトの作成と破棄が、処理時間とメモリー使用率という点でかなりの負担になります。C++ コンパイラは一時オブジェクトの一部を削除しますが、すべてを削除できるとは限りません。

プログラムの明瞭さを保ちつつ、一時オブジェクトの数が最少になるように関数を記述してください。このための手法としては、暗黙の一時オブジェクトに代わって明示的な変数を使用すること、値パラメータに代わって参照パラメータを使用することなどがあります。また、+ と = だけを実装して使用するのではなく、+= のような演算を

実装および使用することもよい手法です。たとえば、次の例の最初の行は、a + b の結果に一時オブジェクトを使用していますが、2 行目は一時オブジェクトを使用していません。

```
T x = a + b;
T x( a ); x += b;
```

インライン関数の使用

小さくて実行速度の速い関数を呼び出す場合は、通常どおりに呼び出すよりもインライン展開する方が効率が上がります。逆に言えば、大きいか実行速度の遅い関数を呼び出す場合は、分岐するよりもインライン展開する方が効率が悪くなります。また、インライン関数の呼び出しはすべて、関数定義が変更されるたびに再コンパイルする必要があります。このため、インライン関数を使用するかどうかは十分な検討が必要です。

関数定義を変更する可能性があり、呼び出し元をすべて再コンパイルするには手間がかかると予測される場合は、インライン関数は使用しないでください。そうでない場合は、関数をインライン展開するコードが関数を呼び出すコードよりも小さいか、あるいはアプリケーションの動作がインライン関数によって大幅に高速化される場合にのみ使用してください。

コンパイラは、すべての関数呼び出しをインライン展開できるわけではありません。そのため、関数のインライン展開の効率を最高にするにはソースを変更しなければならない場合があります。どのような場合に関数がインライン展開されないかを知るには、+w オプションを使用してください。次のような状況では、コンパイラは関数をインライン展開しません。

- ループ、switch 文、try および catch 文のような難しい制御構造が関数に含まれる場合。実際には、これらの関数では、その難しい制御構造はごくまれにしか実行されません。このような関数をインライン展開するには、難しい制御構造が入った内側部分と、内側部分を呼び出すかどうかを決定する外側部分の 2 つに関数を分割します。コンパイラが関数全体をインライン展開できる場合でも、このようによく使用する部分とめったに使用しない部分を分けることで、パフォーマンスを高めることができます。

- インライン関数本体のサイズが大きいか、あるいは複雑な場合。見たところ単純な関数本体は、本体内でほかのインライン関数を呼び出していたり、あるいはコンストラクタやデストラクタを暗黙に呼び出していたりするために複雑な場合があります (派生クラスのコンストラクタとデストラクタでこのような状況がよく起きる)。このような関数ではインライン展開でパフォーマンスが大幅に向上することはめったにないため、インライン展開しないことをお勧めします。
- インライン関数呼び出しの引数が大きいか、あるいは複雑な場合。インラインメンバー関数を呼び出すためのオブジェクトが、そのインライン関数呼び出しの結果である場合は、パフォーマンスが大幅に下がります。複雑な引数を持つ関数をインライン展開するには、その関数引数を局所変数を使用して関数に渡してください。

デフォルト演算子の使用

クラス定義がパラメータのないコンストラクタ、コピーコンストラクタ、コピー代入演算子、またはデストラクタを宣言しない場合、コンパイラがそれらを暗黙的に宣言します。こうして宣言されたものはデフォルト演算子と呼ばれます。C のような構造体は、デフォルト演算子を持っています。デフォルト演算子は、優れたコードを生成するためにどのような作業が必要かを把握しています。この結果作成されるコードは、ユーザーが作成したコードよりもはるかに高速です。これは、プログラマが通常使用できないアセンブリレベルの機能をコンパイラが利用できるためです。そのため、デフォルト演算子が必要な作業をこなしてくれる場合は、プログラムでこれらの演算子をユーザー定義によって宣言する必要はありません。

デフォルト演算子はインライン関数であるため、インライン関数が適切でない場合にはデフォルト演算子を使用しないでください (前の節を参照)。デフォルト演算子は、次のような場合に適切です。

- ユーザーが記述するパラメータのないコンストラクタが、その基底オブジェクトとメンバー変数に対してパラメータのないコンストラクタだけを呼び出す場合。基本の型は、「何も行わない」パラメータのないコンストラクタを効率よく受け入れます。
- ユーザーが記述するコピーコンストラクタが、すべての基底オブジェクトとメンバー変数をコピーする場合
- ユーザーが記述するコピー代入演算子が、すべての基底オブジェクトとメンバー変数をコピーする場合

- ユーザーが記述するデストラクタが空の場合

C++ のプログラミングを紹介する書籍の中には、コードを読んだ際にコードの作成者がデフォルト演算子の効果を考慮に入れていることがわかるように、常にすべての演算子を定義することを勧めているものもあります。しかし、そうすることは明らかに上記で述べた最適化と相入れないものです。デフォルト演算子の使用について明示するには、クラスがデフォルト演算子を使用していることを説明したコメントをコードに入れることをお勧めします。

値クラスの使用

構造体や共用体などの C++ クラスは、値によって渡され、値によって返されます。POD (Plain-Old-Data) クラスの場合、C++ コンパイラは構造体を C コンパイラと同様に渡す必要があります。これらのクラスのオブジェクトは、直接渡されます。ユーザー定義のコピーコンストラクタを持つクラスのオブジェクトの場合、コンパイラは実際にオブジェクトのコピーを構築し、コピーにポインタを渡し、ポインタが戻った後にコピーを破棄する必要があります。これらのクラスのオブジェクトは、間接的に渡されます。この 2 つの条件の中間に位置するクラスの場合は、コンパイラによってどちらの扱いにするかが選択されます。しかし、そうすることでバイナリ互換性に影響が発生するため、コンパイラは各クラスに矛盾が出ないように選択する必要があります。

ほとんどのコンパイラでは、オブジェクトを直接渡すと実行速度が上がります。特に、複素数や確率値のような小さな値クラスの場合に、実行速度が大幅に上がります。そのためプログラムの効率は、間接的ではなく直接渡される可能性が高いクラスを設計することによって向上する場合があります。

互換モード (`-compat[=4]`) では、クラスに次の要素が含まれる場合、クラスは間接的に渡されます。

- ユーザー定義のコンストラクタ
- 仮想関数
- 仮想基底クラス
- 間接的に渡される基底クラス
- 間接的に渡される非静的データメンバー

これらの要素が含まれない場合は、クラスは直接渡されます。

標準モード (デフォルトモード) では、クラスに次の要素が含まれる場合、クラスは間接的に渡されます。

- ユーザー定義のコピーコンストラクタ
- ユーザー定義のデストラクタ
- 間接的に渡される基底クラス
- 間接的に渡される非静的データメンバー

これらの要素が含まれない場合は、クラスは直接渡されます。

クラスを直接渡す

クラスが直接渡される可能性を最大にするには、次のようにしてください。

- 可能な限りデフォルトのコンストラクタ (特にデフォルトのコピーコンストラクタ) を使用する。
- 可能な限りデフォルトのデストラクタを使用する。デフォルトデストラクタは仮想ではないため、デフォルトデストラクタを使用したクラスは、通常は基底クラスにするべきではありません。
- 仮想関数と仮想基底クラスを使用しない。

各種のプロセッサでクラスを直接渡す

C++ コンパイラによって直接渡されるクラス (および共用体) は、C コンパイラが構造体 (または共用体) を渡す場合とまったく同じように渡されます。しかし、C++ の構造体と共用体の渡し方は、アーキテクチャによって異なります。

表 10-1 アーキテクチャ別の構造体と共用体の渡し方

アーキテクチャ	内容の説明
SPARC V7 および V8	構造体と共用体は、呼び出し元で記憶領域を割り当て、ポインタをその記憶領域に渡すことによって渡されます (つまり、構造体と共用体はすべて参照により渡されます)。
SPARC V9	16 バイト (32 バイト) 以下の構造体は、レジスタ中で渡され (返され) ます。共用体と他のすべての構造体は、呼び出し元で記憶領域を割り当て、ポインタをその記憶領域に渡すことによって渡され (返され) ます (つまり、小さな構造体はレジスタ中で渡され、共用体と大きな構造体は参照により渡されます)。この結果、小さな値のクラスは基本の型と同じ効率で渡されることになります。
IA プラットフォーム	構造体と共用体を渡すには、スタックで領域を割り当て、引数をそのスタックにコピーします。構造体と共用体を返すには、呼び出し元のフレームに一時オブジェクトを割り当て、一時オブジェクトのアドレスを暗黙の最初のパラメータとして渡します。

メンバー変数のキャッシュ

C++ メンバー関数では、メンバー変数へのアクセスが頻繁に行われます。

そのため、コンパイラは、`this` ポインタを介してメモリーからメンバー変数を読み込まなければならないことがよくあります。値はポインタを介して読み込まれているため、次の読み込みをいつ行うべきか、あるいは先に読み込まれている値がまだ有効であるかどうかをコンパイラが決定できないことがあります。このような場合、コンパイラは安全な (しかし遅い) 手法を選択し、アクセスのたびにメンバー変数を再読み込みする必要があります。

不要なメモリー再読み込みが行われないようにするには、次のようにメンバー変数の値を局所変数に明示的にキャッシュしてください。

- 局所変数を宣言し、メンバー変数の値を使用して初期化する
- 関数全体で、メンバー変数の代わりに局所変数を使用する
- 局所変数が変わる場合は、局所変数の最終値をメンバー変数に代入する。しかし、メンバー関数がそのオブジェクトの別のメンバー関数を呼び出す場合には、この最適化のために意図しない結果が発生する場合があります。

この最適化は、基本の型の場合と同様に、値をレジスタに置くことができる場合にもっとも効果的です。また、別名の使用が減ることによりコンパイラの最適化が行われやすくなるため、記憶領域を使用する値にも効果があります。

この最適化は、メンバー変数が明示的に、あるいは暗黙的に頻繁に参照渡しされる場合には逆効果になる場合があります。

現在のオブジェクトとメンバー関数の引数の 1 つの間に別名が存在する可能性がある場合などには、クラスの意味を望ましいものにするために、メンバー変数を明示的にキャッシュしなければならないことがあります。例を次に示します。

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

上のコードが次の指令で呼び出されると、意図しない結果になります。

```
x*=x;
```

第11章

マルチスレッドプログラムの構築

この章では、マルチスレッドプログラムの構築方法を説明します。さらに、例外の使用、C++ 標準ライブラリのオブジェクトをスレッド間で共有する方法、従来の (旧形式の) iostream をマルチスレッド環境で使用する方法についても取り上げます。

マルチスレッド処理の詳細については、『マルチスレッドのプログラミング』、『Tools.h++ ユーザーズガイド』、『標準 C++ ライブラリ・ユーザーズガイド』を参照してください。

マルチスレッドプログラムの構築

C++ コンパイラに付属しているライブラリは、すべてマルチスレッドで使用しても安全です。マルチスレッドアプリケーションを作成したい場合や、アプリケーションをマルチスレッド化されたライブラリにリンクしたい場合は、`-mt` オプションを付けてプログラムのコンパイルとリンクを行う必要があります。このオプションを付けると、`-D_REENTRANT` がプリプロセッサに渡され、`-lthread` が `ld` に正しい順番で渡されます。互換性モード (`-compat[=4]`) の場合、`-mt` オプションは `libthread` を `libC` の前にリンクします。標準モード (デフォルトモード) の場合、`-mt` オプションは `libthread` を `libCrun` の前にリンクします。

`-lthread` を使用してアプリケーションを直接リンクしないでください。`libthread` が誤った順番でリンクされます。

マルチスレッドアプリケーションのコンパイルとリンクを別々に行う場合は、次のように入力します。

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

次のように入力すると、マルチスレッドアプリケーションが正しく構築されません。

```
example% CC -c -mt myprog.o
example% CC myprog.o -lthread <- libthread が正しい順番でリンクされない
```

マルチスレッドコンパイルの確認

ldd コマンドを使用すると、アプリケーションが libthread にリンクされたかどうかを確認することができます。

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>      /usr/lib/libm.so.1
libCrun.so.1 =>   /usr/lib/libCrun.so.1
libw.so.1 =>      /usr/lib/libw.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

C++ サポートライブラリの使用

C++ サポートライブラリ (libCrun、libiostream、libCstd、libC) は、マルチスレッドで使用しても安全ですが、非同期安全 (非同期例外で使用しても安全) ではありません。したがって、マルチスレッドアプリケーションのシグナルハンドラでは、これらのライブラリに含まれている関数を使用しないでください。使用するとデッドロックが発生する可能性があります。

マルチスレッドアプリケーションのシグナルハンドラでは、次のものは安全に使用できません。

- iostream
- new 式と delete 式
- 例外

マルチスレッドプログラムでの例外の使用

現在実装されている例外処理は、マルチスレッドで使用しても安全です。すなわち、あるスレッドの例外によって、別のスレッドの例外が阻害されることはありません。ただし、例外を使用して、スレッド間で情報を受け渡すことはできません。すなわち、あるスレッドから送出された例外を、別のスレッドで捕獲することはできません。

それぞれのスレッドでは、独自の `terminate()` 関数と `unexpected()` 関数を設定することができます。あるスレッドで呼び出した `set_terminate()` 関数や `set_unexpected()` 関数は、そのスレッドの例外だけに影響します。デフォルトの `terminate()` 関数の内容は、メインスレッドでは `abort()` になり、それ以外のスレッドでは `thr_exit()` になります (103 ページの「実行時エラーの指定」) を参照してください)。

注 – スレッドの取り消し (`pthread_cancel(3T)`) を行うと、スタック上の自動オブジェクト (静的ではない局所オブジェクト) が破棄されます。スレッドが取り消されると、局所デストラクタの実行中に、ユーザーが `pthread_cleanup_push()` を使用して登録したクリーンアップルーチンが実行されます。クリーンアップルーチンの登録後に呼び出した関数の局所オブジェクトは、そのクリーンアップルーチンが実行される前に破棄されます。

C++ 標準ライブラリのオブジェクトのスレッド間での共有

C++ 標準ライブラリ (`libCstd`) は、マルチスレッドで使用しても安全です。すなわち、このライブラリの内部は、マルチスレッド環境で正しく機能します。ただし、このライブラリのオブジェクトのうち、プログラム自身がスレッド間で共有するものについては、`iostream` オブジェクトと `locale` オブジェクトを除いて、プログラム自身による明示的なロックが必要です。

たとえば、文字列をインスタンス化し、この文字列を新しく生成したスレッドに参照で渡した場合を考えてみましょう。この文字列への書き込みアクセスはロックする必要があります。なぜなら、同じ文字列オブジェクトを、プログラムが複数のスレッドで明示的に共有しているからです (この処理を行うために用意された C++ 標準ライブラリの機能については後述します)。

これに対して、この文字列を新しいスレッドに値で渡した場合は、ロックについて考慮する必要はありません。このことは、Rogue Wave の「書き込み時コピー」機能により、2 つのスレッドの別々の文字列が同じ表現を共有している場合にも当てはまります。このような場合のロックは、ライブラリが自動的に処理します。プログラム自身でロックを行う必要があるのは、スレッド間での参照渡しや、大域オブジェクトや静的オブジェクトを使用して、同じオブジェクトを複数のスレッドから明示的に使用できるようにした場合だけです。

ここからは、複数のスレッドが存在する場合の動作を保証するために、C++ 標準ライブラリの内部で使用されるロック (同期) 機能について説明します。

マルチスレッドでの安全性を実現する機能は、2 つの同期クラス、`_RWSTDMutex` と `_RWSTDGuard` によって提供されます。

`_RWSTDMutex` クラスは、プラットフォームに依存しないロック機能を提供します。このクラスには、次のメンバー関数があります。

- `void acquire()`: 自分自身に対するロックを獲得する。または、このロックを獲得できるまでブロックする。
- `void release()`: 自分自身に対するロックを解除する。

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

`_RWSTDGuard` クラスは、`_RWSTDMutex` クラスのオブジェクトをカプセル化するための便利なラッパークラスです。`_RWSTDGuard` クラスのオブジェクトは、自分自身のコンストラクタの中で、カプセル化された相互排他ロック (mutex) を獲得しようとします (エラーが発生した場合は、このコンストラクタは `std::exception` から派生

している ::thread_error 型の例外を送出します)。獲得された相互排他ロックは、このオブジェクトのデストラクタの中で解除されます (このデストラクタは例外を送出しません)。

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

さらに、_RWSTD_MT_GUARD(mutex) マクロ (従来の _STDGUARD) を使用すると、マルチスレッドの構築時にだけ _RWSTDGuard クラスのオブジェクトを生成することができます。生成されたオブジェクトは、そのオブジェクトが定義されたコードブロックの残りの部分が、複数のスレッドで同時に実行されないようにします。単一スレッドの構築時には、このマクロは空白の式に展開されます。

これらの機能は、次のように使用します。

```
#include <rw/stdmutex.h>

//
// 複数のスレッドで共有する整数
//
int I;

//
// I の更新の同期をとるために使用する相互排他ロック (mutex)
//
_RWSTDMutex I_mutex;

//
// I を 1 だけ増分する。 Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire();  // mutex をロック
   I++;
   I_mutex.release();  // mutex のロックを解除
}

//
// I を 1 だけ減分する。 Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex);  // I_mutex のロックを獲得
   --I;
   //
   // I のロックは guard のデストラクタが呼び出されたときに解除される
   //
}
```

マルチスレッド環境での従来の iostream の使用

この節では、libC ライブラリと libiostream ライブラリの iostream クラスを、マルチスレッド環境での入出力に使用する方法を説明します。さらに、iostream クラスの派生クラスを作成し、ライブラリの機能を拡張する例も紹介します。ここでは、C++ のマルチスレッドコードを記述するための指針は示しません。

この節では、従来の iostream (libC と libiostream) だけを取り扱います。この節の説明は、C++ 標準ライブラリに含まれている新しい iostream (libCstd) には当てはまりません。

iostream ライブラリのインタフェースは、マルチスレッド環境用のアプリケーション、すなわちバージョン 2.6、7、8 の Solaris オペレーティング環境で実行されたときにマルチスレッド機能を使用するプログラムから使用することができます。従来のライブラリのシングルスレッド機能を使用するアプリケーションは影響を受けません。

ライブラリが「マルチスレッドを使用しても安全」といえるのは、複数のスレッドが存在する環境で正しく機能する場合です。一般に、ここでの「正しく機能する」とは、公開関数がすべて再入可能なことを指します。iostream ライブラリには、複数のスレッドの間で共有されるオブジェクト (C++ クラスのインスタンス) の状態が、複数のスレッドから変更されるのを防ぐ機能があります。ただし、iostream オブジェクトがマルチスレッドで使用しても安全になるのは、そのオブジェクトの公開メンバー関数が実行されている間に限られます。

注 – アプリケーションで libC ライブラリのマルチスレッドで使用しても安全なオブジェクトを使用しているからといって、そのアプリケーションが自動的にマルチスレッドで使用しても安全になるわけではありません。アプリケーションがマルチスレッドで使用しても安全になるのは、マルチスレッド環境で想定したとおりに実行される場合だけです。

マルチスレッドで使用しても安全な iostream ライブラリの構成

マルチスレッドで使用しても安全な iostream ライブラリの構成は、従来の iostream ライブラリの構成と多少異なります。マルチスレッドで使用しても安全な iostream ライブラリのインタフェースは、iostream クラスやその基底クラスの公

開および限定公開のメンバー関数を示していて、従来のライブラリと整合性が保たれていますが、クラス階層に違いがあります。詳細については、141 ページの「iostream ライブラリのインタフェースの変更」 を参照してください。

従来の中核クラスの名前が変更されています (先頭に unsafe_ という文字列が付きました)。iostream パッケージの中核クラスを表 11-1 に示します。

表 11-1 iostream の中核クラス

クラス	内容
stream_MT	マルチスレッドで使用しても安全なクラスの基底クラス
streambuf	バッファーの基底クラス
unsafe_ios	各種のストリームクラスに共通の状態変数 (エラー状態、書式状態など) を収容するクラス
unsafe_istream	streambuf から取り出した文字の並びを、書式付き/書式なし変換する機能を持つクラス
unsafe_ostream	streambuf に格納する文字の並びを、書式付き/書式なし変換する機能を持つクラス
unsafe_iostream	unsafe_istream クラスと unsafe_ostream クラスを組み合わせた入出力兼用のクラス

マルチスレッドで使用しても安全なクラスは、すべて基底クラス stream_MT の派生クラスです。また、これらのクラスは、streambuf を除いて、(先頭に unsafe_ が付いた) 従来の基底クラスの派生クラスでもあります。この例を次に示します。

```
class streambuf: public stream_MT { ... };
class ios: virtual public unsafe_ios, public stream_MT { ... };
class istream: virtual public ios, public unsafe_istream { ... };
```

stream_MT には、それぞれの iostream クラスをマルチスレッドで使用しても安全にするための相互排他 (mutex) ロック機能が含まれています。また、このクラスには、マルチスレッドで使用しても安全な属性を動的に変更できるように、ロックを動的に有効および無効にする機能もあります。入出力変換とバッファー管理の基本機能は、従来の unsafe_ クラスにまとめられています。したがって、ライブラリに新しく追加されたマルチスレッドで使用しても安全な機能は、その派生クラスだけで使用することができます。マルチスレッドで使用しても安全なクラスには、従来の unsafe_ 基底クラスと同じ公開メンバー関数と限定公開メンバー関数が含まれていま

す。これらのメンバー関数は、オブジェクトをロックし、unsafe_ 基底クラスの同名の関数を呼び出し、その後でオブジェクトのロックを解除するラッパーとして働きます。

注 – streambuf クラスは、unsafe_ クラスの派生クラスではありません。streambuf クラスの公開メンバー関数と限定公開メンバー関数は、ロックを行うことで再入可能になります。ロックを行わない関数も用意されています。これらの関数は、名前の後ろに _unlocked という文字列が付きます。

公開変換ルーチン

iostream のインタフェースには、マルチスレッドで使用しても安全な、再入可能な公開関数が追加されています。これらの関数は、追加引数としてユーザーが指定したバッファーを受け取ります。これらの関数を以下に示します。

表 11-2 マルチスレッドで使用しても安全な、再入可能な公開関数

関数	内容
`char *oct_r (char *buf, int buflen, long num, int width)`	数値を 8 進数の形式で表現した ASCII 文字列のポインタを返す。width が 0 (ゼロ) ではない場合は、その値が書式設定用のフィールド幅になります。戻り値は、ユーザーが用意したバッファの先頭を指すとは限りません。
`char *hex_r (char *buf, int buflen, long num, int width)`	数値を 16 進数の形式で表現した ASCII 文字列のポインタを返す。width が 0 (ゼロ) ではない場合は、その値が書式設定用のフィールド幅になります。戻り値は、ユーザーが用意したバッファの先頭を指すとは限りません。

表 11-2 マルチスレッドで使用しても安全な、再入可能な公開関数 (続き)

関数	内容
`char *dec_r (char *buf, int buflen, long num, int width)`	数値を 10 進数の形式で表現した ASCII 文字列のポインタを返す。width が 0 (ゼロ) ではない場合は、その値が書式設定用のフィールド幅になります。戻り値は、ユーザーが用意したバッファの先頭を指すとは限りません。
`char *chr_r (char *buf, int buflen, long num, int width)`	文字 chr を含む ASCII 文字列のポインタを返す。width が 0 (ゼロ) ではない場合は、その値と同じ数の空白に続けて chr が格納されます。戻り値は、ユーザーが用意したバッファの先頭を指すとは限りません。
`char *form_r (char *buf, int buflen, long num, int width)`	sprintf によって書式設定した文字列のポインタを返す (書式文字列 format 以降のすべての引数を使用)。ユーザーが用意したバッファに、変換後の文字列を収容できるだけの大きさがなければなりません。

注 – 従来の libC との互換性を確保するために提供されている iostream ライブラリの公開変換ルーチン (oct、hex、dec、chr、form) は、マルチスレッドで使用すると安全ではありません。

マルチスレッドで使用しても安全な libC ライブラリを使用したコンパイルとリンク

libC ライブラリの iostream クラスを使用した、マルチスレッド環境用のアプリケーションを構築するには、-mt オプションを付けてソースコードのコンパイルとリンクを行う必要があります。このオプションを付けると、プリプロセッサに -D_REENTRANT が渡され、リンカーに -lthread が渡されます。

注 – libC と libthread へのリンクを行うには、(-lthread オプションではなく) -mt オプションを使用します。このオプションを使用しないと、ライブラリが正しい順番でリンクされないことがあります。誤って-lthread オプションを使用すると、作成したアプリケーションが正しく機能しない場合があります。

iostream クラスを使用するシングルスレッドアプリケーションについては、コンパイラオプションやリンカオプションは特に必要ありません。オプションを何も指定しなかった場合は、コンパイラは libC ライブラリへのリンクを行います。

マルチスレッドで使用しても安全な iostream の制約

iostream ライブラリのマルチスレッドでの安全性には制約があります。これは、マルチスレッド環境で iostream オブジェクトが共有された場合に、iostream を使用するプログラミング手法の多くが安全ではなくなるためです。

エラー状態のチェック

マルチスレッドでの安全性を実現するには、エラーの原因になる入出力操作を含んでいる危険領域で、エラーチェックを行う必要があります。エラーが発生したかどうかを確認するには次のようにします。

コード例 11-1 エラー状態のチェック

```
#include <iostream.h>
enum iostate { IOok, IOeof, IOfail };

iostate read_number(istream& istr, int& num)
{
    stream_locker sl(istr, stream_locker::lock_now);

    istr >> num;

    if (istr.eof()) return IOeof;
    if (istr.fail()) return IOfail;
    return IOok;
}
```

この例では、stream_locker オブジェクト sl のコンストラクタによって、istream オブジェクト istr がロックされます。このロックは、read_number が終了したときに呼び出される sl のデストラクタによって解除されます。

最後の書式なし入力操作で抽出された文字列の取得

マルチスレッドでの安全性を実現するには、最後の入力操作と gcount の呼び出しを行う期間に、istream オブジェクトを排他的に使用するスレッドの内部から、gcount 関数を呼び出す必要があります。gcount は次のように呼び出します。

コード例 11-2 gcount の呼び出し

```
#include <iostream.h>
#include <rlocks.h>
void fetch_line(istream& istr, char* line, int& linecount)
{
    stream_locker sl(istr, stream_locker::lock_defer);

    sl.lock(); // lock the stream istr
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // unlock istr
    ...
}
```

この例では、stream_locker クラスの lock メンバー関数を呼び出してから unlock メンバー関数を呼び出すまでが、プログラムの相互排他領域になります。

ユーザー定義の入出力操作

マルチスレッドでの安全性を実現するには、別々の操作を特定の順番で行う必要があるユーザー定義型用の入出力操作を、危険領域としてロックする必要があります。この入出力操作の例を以下に示します。

コード例 11-3 ユーザー定義の入出力操作

```
#include <rlocks.h>
#include <iostream.h>
class mystream: public istream {

    // その他の定義 ...
    int getRecord(char* name, int& id, float& gpa);
};

int mystream::getRecord(char* name, int& id, float& gpa)
{
    stream_locker sl(this, stream_locker::lock_now);
```

コード例 11-3 ユーザー定義の入出力操作 (続き)

```
    *this >> name;
    *this >> id;
    *this >> gpa;

    return this->fail() == 0;
}
```

マルチスレッドで使用しても安全なクラスのパフォーマンスオーバーヘッドの削減

現行の libC ライブラリに含まれているマルチスレッドで使用しても安全なクラスを使用すると、シングルスレッドアプリケーションの場合でさえも多少のオーバーヘッドが発生します。libC の unsafe_ クラスを使用すると、このオーバーヘッドを回避することができます。

次のようにスコープ決定演算子を使用すると、unsafe_ 基底クラスのメンバー関数を実行することができます。

```
    cout.unsafe_ostream::put('4');
```

```
    cin.unsafe_istream::read(buf, len);
```

注 – unsafe_ クラスは、マルチスレッドアプリケーションでは安全に使用できません。

unsafe_ クラスを使用する代わりに、cout オブジェクトと cin オブジェクトを unsafe にしてから、通常の操作を行うこともできます。ただし、パフォーマンスが若干低下します。unsafe な cout と cin は、次のように使用します。

コード例 11-4 マルチスレッドでの安全性の無効化

```
#include <iostream.h>
//マルチスレッドでの安全性を無効化
cout.set_safe_flag(stream_MT::unsafe_object);
```

コード例 11-4 マルチスレッドでの安全性の無効化 (続き)

```
//マルチスレッドでの安全性を無効化
cin.set_safe_flag(stream_MT::unsafe_object);
cout.put('4');
cin.read(buf, len);
```

iostream オブジェクトがマルチスレッドで使用しても安全な場合は、相互排他ロックを行うことで、そのオブジェクトのメンバー変数が保護されます。しかし、シングルスレッド環境でしか実行されないアプリケーションでは、このロック処理のために、本来なら必要のないオーバーヘッドがかかります。iostream オブジェクトのマルチスレッドでの安全性の有効/無効を動的に切り替えると、パフォーマンスを改善することができます。たとえば、iostream オブジェクトのマルチスレッドでの安全性を無効にするには、次のようにします。

コード例 11-5 マルチスレッドでの安全性の無効化

```
fs.set_safe_flag(stream_MT::unsafe_object);// マルチスレッドでの安
全性を無効化
    .... do various i/o operations
```

iostream が複数のスレッド間で共有されないコード領域では、マルチスレッドでの安全性の無効化ストリームであっても安全に使用することができます。たとえば、スレッドが 1 つしかないプログラムや、スレッドごとに非公開の iostream を使用するプログラムでは問題は起きません。

プログラムに同期処理を明示的に挿入すると、iostream が複数のスレッド間で共有される場合にも、マルチスレッドで使用すると安全ではない iostream を安全に使用できるようになります。この例を次に示します。

コード例 11-6 マルチスレッドで使用すると安全ではないオブジェクトの同期処理

```
    generic_lock() ;
    fs.set_safe_flag(stream_MT::unsafe_object) ;
    ... do various i/o operations
    generic_unlock() ;
```

ここで、generic_lock 関数と generic_unlock 関数は、相互排他ロック (mutex)、セマフォ、読み取り/書き込みロックといった基本型を使用する同期機能であれば、何でもかまいません。

注 – この目的のためには、libC ライブラリの stream_locker クラスを使用すると便利です。

詳細については、145 ページの「オブジェクトのロック」 を参照してください。

iostream ライブラリのインタフェースの変更

この節では、iostream ライブラリをマルチスレッドで使用しても安全にするために行われたインタフェースの変更内容について説明します。

新しいクラス

libC インタフェースに追加された新しいクラスを次の表に示します。

コード例 11-7 新しいクラス

```
    stream_MT
    stream_locker
    unsafe_ios
    unsafe_istream
    unsafe_ostream
    unsafe_iostream
    unsafe_fstreambase
    unsafe_strstreambase
```

新しいクラス階層

iostream インタフェースに追加された新しいクラス階層を次の表に示します。

コード例 11-8 新しいクラス階層

```
class streambuf : public stream_MT { ... };
class unsafe_ios { ... };
class ios : virtual public unsafe_ios, public stream_MT  { ... };
class unsafe_fstreambase : virtual public unsafe_ios { ... };
class fstreambase : virtual public ios, public unsafe_fstreambase
  { ... };
class unsafe_strstreambase : virtual public unsafe_ios { ... };
class strstreambase : virtual public ios, public
unsafe_strstreambase { ... };
```

コード例 11-8 新しいクラス階層 (続き)

```
class unsafe_istream : virtual public unsafe_ios  { ... };
class unsafe_ostream : virtual public unsafe_ios  { ... };
class istream : virtual public ios, public unsafe_istream { ... };
class ostream : virtual public ios, public unsafe_ostream { ... };
class unsafe_iostream : public unsafe_istream, public unsafe_ostream
{ ... };
```

新しい関数

iostream インタフェースに追加された新しい関数を次の表に示します。

コード例 11-9 新しい関数

```
class streambuf {
public:
  int sgetc_unlocked();
  void sgetn_unlocked(char *, int);
  int snextc_unlocked();
  int sbumpc_unlocked();
  void stossc_unlocked();
  int in_avail_unlocked();
  int sputbackc_unlocked(char);
  int sputc_unlocked(int);
  int sputn_unlocked(const char *, int);
  int out_waiting_unlocked();
protected:
  char* base_unlocked();
  char* ebuf_unlocked();
  int blen_unlocked();
  char* pbase_unlocked();
  char* eback_unlocked();
  char* gptr_unlocked();
  char* egptr_unlocked();
  char* pptr_unlocked();
  void setp_unlocked(char*, char*);
  void setg_unlocked(char*, char*, char*);
  void pbump_unlocked(int);
  void gbump_unlocked(int);
  void setb_unlocked(char*, char*, int);
  int unbuffered_unlocked();
  char *epptr_unlocked();
  void unbuffered_unlocked(int);
  int allocate_unlocked(int);
```

コード例 11-9 新しい関数 (続き)

```
};

class filebuf : public streambuf {
public:
 int is_open_unlocked();
 filebuf* close_unlocked();
 filebuf* open_unlocked(const char*, int, int =
   filebuf::openprot);

 filebuf* attach_unlocked(int);
};

class strstreambuf : public streambuf {
public:
 int freeze_unlocked();
 char* str_unlocked();
};


unsafe_ostream& endl(unsafe_ostream&);
unsafe_ostream& ends(unsafe_ostream&);
unsafe_ostream& flush(unsafe_ostream&);
unsafe_istream& ws(unsafe_istream&);
unsafe_ios& dec(unsafe_ios&);
unsafe_ios& hex(unsafe_ios&);
unsafe_ios& oct(unsafe_ios&);

char* dec_r (char* buf, int buflen, long num, int width)
char* hex_r (char* buf, int buflen, long num, int width)
char* oct_r (char* buf, int buflen, long num, int width)
char* chr_r (char* buf, int buflen, long chr, int width)
char* str_r (char* buf, int buflen, const char* format, int width
   = 0);
char* form_r (char* buf, int buflen, const char* format, ...)
```

大域データと静的データ

マルチスレッドアプリケーションでの大域データと静的データは、スレッド間で安全に共有されません。スレッドはそれぞれ個別に実行されますが、同じプロセス内のスレッドは、大域オブジェクトと静的オブジェクトへのアクセスを共有します。このような共有オブジェクトをあるスレッドで変更すると、その変更が同じプロセス内の他のスレッドにも反映されるため、状態を保つことが難しくなります。C++ では、クラ

スオブジェクト (クラスのインスタンス) の状態は、メンバー変数の値が変わると変化します。そのため、共有されたクラスオブジェクトは、他のスレッドからの変更に対して脆弱です。

マルチスレッドアプリケーションで iostream: ライブラリを使用し、iostream.h をインクルードすると、デフォルトでは標準ストリーム (cout、cin、cerr、clog) が大域的な共有オブジェクトとして定義されます。iostream ライブラリはマルチスレッドで使用しても安全なので、iostream オブジェクトのメンバー関数の実行中は、共有オブジェクトの状態が、他のスレッドからのアクセスや変更から保護されます。ただし、オブジェクトがマルチスレッドで使用しても安全なのは、そのオブジェクトの公開メンバー関数が実行されている間だけです。そのためには、次のような方法があります。

```
int c;
cin.get(c);
```

このコードを使用して、スレッド A が get バッファーの次の文字を取り出し、バッファーポインタを更新したとします。ところが、スレッド A が、次の命令で再び get を呼び出したとしても、シーケンスのその次の文字が返される保証はありません。なぜなら、スレッド A の 2 つの get の呼び出しの間に、スレッド B からも別の get が呼び出される可能性があるからです。

このような共有オブジェクトとマルチスレッド処理の問題に対処する方法については、145 ページの「オブジェクトのロック」を参照してください。

連続実行

iostream オブジェクトを使用した場合に、一続きの入出力操作をマルチスレッドで使用しても安全にしなければならない場合がよくあります。たとえば、次のコードを考えてみましょう。

```
cout << " Error message:" << errstring[err_number] << "\n";
```

このコードでは、cout ストリームオブジェクトの 3 つのメンバー関数が実行されます。cout は共有オブジェクトなので、マルチスレッド環境では、この操作全体を危険領域として不可分的に (連続して) 実行しなければなりません。iostream クラスのオブジェクトに対する一続きの操作を不可分的に実行するには、何らかのロック処理が必要です。

iostream オブジェクトをロックできるように、libC ライブラリに新しく stream_locker クラスが追加されています。このクラスの詳細については、145 ページの「オブジェクトのロック」を参照してください。

オブジェクトのロック

共有オブジェクトとマルチスレッド処理の問題に対処するもっとも簡単な方法は、iostream オブジェクトをスレッドの局所的なオブジェクトにして、問題そのものを解消してしまうことです。そのためには、次のような方法があります。

- スレッドのエントリ関数の中でオブジェクトを局所的に宣言する。
- スレッド固有データの中でオブジェクトを宣言する (スレッド固有データの使用法については、thr_keycreate(3T) のマニュアルページを参照してください)。
- ストリームオブジェクトを特定のスレッド専用にする。このオブジェクトスレッドは、慣例により非公開 (private) になります。

ただし、デフォルトの共有標準ストリームオブジェクトを初めとして、多くの場合はオブジェクトをスレッドの局所的なオブジェクトにすることはできません。そのため、別の手段が必要です。

iostream クラスのオブジェクトに対する一続きの操作を不可分的に実行するには、何らかのロック処理が必要です。ただし、ロック処理を行うと、シングルスレッドアプリケーションの場合でさえも、オーバーヘッドが多少増加します。ロック処理を追加する必要があるか、それとも iostream オブジェクトをスレッドの非公開オブジェクトにすればよいかは、アプリケーションで採用しているスレッドモデル (独立スレッドと連携スレッドのどちらを使用しているか) によって決まります。

- スレッドごとに別々の iostream オブジェクトを使用してデータを入出力する場合は、それぞれの iostream オブジェクトが、該当するスレッドの非公開オブジェクトになります。ロック処理の必要はありません。
- 複数のスレッドを連携させる (これらのスレッドの間で、同じ iostream オブジェクトを共有させる) 場合は、その共有オブジェクトへのアクセスの同期をとる必要があり、何らかのロック処理によって、一続きの操作を不可分的にする必要があります。

stream_locker クラス

iostream ライブラリには、iostream オブジェクトに対する一続きの操作をロックするための stream_locker クラスが含まれています。これにより、iostream オブジェクトのロックを動的に切り換えることで生じるオーバーヘッドを最小限にすることができます。

stream_locker クラスのオブジェクトを使用すると、ストリームオブジェクトに対する一続きの操作を不可分的にすることができます。たとえば、次の例を考えてみましょう。このコードは、ファイル内の位置を特定の場所まで移動し、その後続のデータブロックを読み込みます。

コード例 11-10 ロック処理の使用例

```
#include <fstream.h>
#include <rlocks.h>

void lock_example (fstream& fs)
{
    const int len = 128;
    char buf[len];
    int offset = 48;
    stream_locker s_lock(fs, stream_locker::lock_now);
    . . . . . .// ファイルを開く
    fs.seekg(offset, ios::beg);
    fs.read(buf, len);
}
```

この例では、stream_locker オブジェクトのコンストラクタが実行されてから、デストラクタが実行されるまでが、一度に 1 つのスレッドしか実行できない相互排他領域になります (デストラクタは、lock_example 関数が終了したときに呼び出されます)。この stream_locker オブジェクトにより、ファイル内の特定のオフセットへの移動と、ファイルからの読み込みの連続的な (不可分的な) 実行が保証され、ファイルからの読み込みを行う前に、別のスレッドによってオフセットが変更されてしまう可能性がなくなります。

stream_locker オブジェクトを使用して、相互排他領域を明示的に定義することもできます。次の例では、入出力操作と、その後で行うエラーチェックを不可分的にするために、stream_locker オブジェクトのメンバー関数、lock と unlock を呼び出しています。

コード例 11-11 入出力操作とエラーチェックの不可分化

```
{
    ...
    stream_locker file_lck(openfile_stream,
                           stream_locker::lock_defer);
    ....
    file_lck.lock();  // openfile_stream をロック
    openfile_stream << "Value: " << int_value << "\n";
    if(!openfile_stream) {
        file_error("Output of value failed\n");
        return;
    }
    file_lck.unlock();  // openfile_stream のロックを解除
}
```

詳細については、stream_locker(3CC4) のマニュアルページを参照してください。

マルチスレッドで使用しても安全なクラス

iostream クラスから新しいクラスを派生させて、機能を拡張または特殊化することができます。マルチスレッド環境で、これらの派生クラスからインスタンス化したオブジェクトを使用する場合は、その派生クラスがマルチスレッドで使用しても安全でなければなりません。

マルチスレッドで使用しても安全なクラスを派生させる場合は、次のことに注意する必要があります。

- クラスオブジェクトの内部状態を複数のスレッドによる変更から保護し、そのオブジェクトをマルチスレッドで使用しても安全にします。そのためには、公開および限定公開のメンバー関数に含まれているメンバー変数へのアクセスを、相互排他ロックで直列化します。
- マルチスレッドで使用しても安全な基底クラスのメンバー関数を、一続きに呼び出す必要がある場合は、それらの呼び出しを stream_locker オブジェクトを使用して不可分にします。

- stream_locker オブジェクトで定義した危険領域の内部では、streambuf クラスの _unlocked メンバー関数を使用して、ロック処理のオーバーヘッドを防止します。
- streambuf クラスの公開仮想関数を、アプリケーションから直接呼び出す場合は、それらの関数をロックします。該当する関数は、xsgetn、underflow、pbackfail、xsputn、overflow、seekoff、seekpos です。
- ios クラスの iword メンバー関数と pword メンバー関数を使用して、ios オブジェクトの書式設定状態を拡張します。ただし、複数のスレッドが iword 関数や pword 関数の同じ添字を共有している場合は、問題が発生することがあります。これらのスレッドをマルチスレッドで使用しても安全にするには、適切なロック機能を使用する必要があります。
- メンバー関数のうち、char 型よりも大きなサイズのメンバー変数値を返すものをロックします。

オブジェクトの破棄

複数のスレッドの間で共有される iostream オブジェクトを削除するには、サブスレッドがそのオブジェクトの使用を終えていることを、メインスレッドで確認する必要があります。共有オブジェクトを安全に破棄する方法を以下に示します。

コード例 11-12 共有オブジェクトの破棄

```
#include <fstream.h>
#include <thread.h>
fstream* fp;

void *process_rtn(void*)
{
    // fp を使用するサブスレッドの本体...
}

void multi_process(const char* filename, int numthreads)
{
    fp = new fstream(filename, ios::in); // fstream オブジェクトを
生成
                                         // スレッドを生成する前に
    // スレッドを生成
    for (int i=0; i<numthreads; i++)
        thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);
```

コード例 11-12 共有オブジェクトの破棄 (続き)

```
    ...
    // スレッドが終了するまで待機
    for (int i=0; i<numthreads; i++)
        thr_join(0, 0, 0);

    delete fp;                          // 後で fstream オブジェクトを削除
    fp = NULL;                              // すべてのスレッドが終了した
}
```

アプリケーションの例

ここでは、libC ライブラリの iostream オブジェクトを安全な方法で使用するマルチスレッドアプリケーションの例を示します。

このアプリケーションは、最大で 255 のスレッドを生成します。それぞれのスレッドは、別々の入力ファイルを 1 行ずつ読み込み、標準出力ストリーム cout を介して共通の出力ファイルに書き出します。この出力ファイルは、すべてのスレッドから共有されるため、出力操作がどのスレッドから行われたかを示す値をタグとして付けます。

コード例 11-13 iostreamオブジェクトをマルチスレッドで使用しても安全な方法で使用

```
// タグ付きスレッドデータの生成
// 出力ファイルに次の形式で文字列を書き出す
//   <タグ><データ文字列>\n
// ここで、<タグ> は unsigned char 型の整数値
// このアプリケーションで最大 255 のスレッドを実行可能
// <データ文字列> は任意のプリント可能文字列
//<タグ> は char 型として書き出される整数値なので、
// 出力ファイルの内容を参照するには、次のように od を使用する
//      od -c out.file |more

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <thread.h>

struct thread_args {
  char* filename;
  int thread_tag;
```

コード例 11-13　iostreamオブジェクトをマルチスレッドで使用しても安全な方法で使用 (続き)

```
};

const int thread_bufsize = 256;

// それぞれのスレッドのエントリルーチン
void* ThreadDuties(void* v) {
// このスレッドの引数を取得
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // 入力ファイルを開く
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // 1 行ずつ読み込む
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // cout ストリームをロックし、入出力操作を不可分にする
    lockout.lock();
  // 行にタグを付けて cout に送出する
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  1 + 各スレッドのファイル名リスト
   if(argc < 2) {
     cout << &dlq;usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// thread_id の配列
  thread_t created_threads[thread_bufsize];
// スレッドのエントリ配列に渡す引数配列
  thread_args thr_args[thread_bufsize];
  int i;
  for( i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// スレッドにタグを割り当てる (255 以下の値)
```

コード例 11-13　iostreamオブジェクトをマルチスレッドで使用しても安全な方法で使用 (続き)

```
    thr_args[i].thread_tag = total_tags++;
// スレッドを生成する
    thr_create(0, 0, ThreadDuties, &thr_args[i],
           THR_SUSPENDED, &created_threads[i]);
  }

  for(i = 0; i < num_threads; i++) {
    thr_continue(created_threads[i]);
  }
  for(i = 0; i < num_threads; i++) {
    thr_join(created_threads[i], 0, 0);
  }
  return 0;
}
```

PART III　ライブラリ

第12章

ライブラリの使用

ライブラリを使用すると、アプリケーション間でコードを共有したり、非常に大規模なアプリケーションを単純化することができます。C++ コンパイラでは、さまざまなライブラリを使用できます。この章では、これらのライブラリの使用方法を説明します。

C ライブラリ

Solaris オペレーティング環境では、いくつかのライブラリが /usr/lib にインストールされます。このライブラリのほとんどは C インタフェースを持っています。デフォルトでは libc、libm、libw ライブラリが CC ドライバによってリンクされます。ライブラリlibthread は、-mt オプションを指定した場合にのみリンクされます。それ以外のシステムライブラリをリンクするには、-l オプションでリンク時に指定する必要があります。たとえば、-libdemangle ライブラリをリンクするには、リンク時に -ldemangle を CC コマンド行に指定します。

```
example% CC text.c -ldemangle
```

C++ コンパイラには、独自の実行時ライブラリが複数あります。すべての C++ アプリケーションは、CC ドライバによってこれらのライブラリとリンクされます。C++ コンパイラには、次の節に示すようにこれ以外にも便利なライブラリがいくつかあります。

C++ コンパイラ付属のライブラリ

Sun C++ コンパイラには、いくつかのライブラリが添付されています。これらのライブラリには、互換モード (`-compat=4`) だけで使用できるもの、標準モード (`-compat=5`) だけで使用できるもの、あるいは両方のモードで使用できるものがあります。`libgc` ライブラリと `libdemangle` ライブラリには C インタフェースがあり、どちらのモードでもアプリケーションにリンクできます。

次の表に、Sun C++ コンパイラに添付されるライブラリと、それらを使用できるモードを示します。

表 12-1 C++ コンパイラに添付されるライブラリ

ライブラリ	内容の説明	使用できるモード
`libstlport`	標準ライブラリの STLport 実装	`-compat=5`
`libstlport_dbg`	デバッグモード用 STLport ライブラリ	`-compat=5`
`libCrun`	C++ 実行時	`-compat=5`
`libCstd`	C++ 標準ライブラリ	`-compat=5`
`libiostream`	従来の `iostream`	`-compat=5`
`libC`	C++ 実行時、従来の `iostream`	`-compat=4`
`libcomplex`	`complex` ライブラリ	`-compat=4`
`librwtool`	`Tools.h++` 7	`-compat=4`, `-compat=5`
`librwtool_dbg`	デバッグ可能な `Tools.h++`7	`-compat=4,-compat=5`
`libgc`	ガベージコレクション	C インタフェース
`libdemangle`	復号化	C インタフェース

注 – STLport、Rogue Wave、または Sun Microsystems C++ ライブラリの構成マクロを再定義したり変更したりしないでください。ライブラリは C++ コンパイラとともに動作するよう構成および構築されています。. libCstd と Tool.h++ は互いに働き合うように構成されているので、その構成マクロを変更すると、プログラムのコンパイルやリンクが行われなくなったり、プログラムが正しく実行されなくなったりします。

C++ライブラリの説明

これらのライブラリについて簡単に説明します。

- `libCrun`:このライブラリには、コンパイラが標準モード (`-compat=5`) で必要とする実行時サポートが含まれています。`new` と `delete`、例外、RTTI がサポートされます。

`libCstd`:これは C++ 標準ライブラリです。特に、このライブラリには `iostream` が含まれています。既存のソースで従来の `iostream` を使用している場合には、ソースを新しいインタフェースに合わせて修正しないと、標準 `iostream` を使用できません。詳細は、オンラインマニュアルの『Standard C++ Library Class Reference』を参照してください。このマニュアルにアクセスするには、Web ブラウザで次のアドレスにアクセスしてください。

```
file:/opt/SUNWspro/docs/index.html
```

コンパイラソフトウェアが `/opt` ディレクトリにインストールされていない場合は、システム上でこのディレクトリに相当するパスをシステム管理者に問い合わせてください。

- `libiostream`:これは標準モード (`-compat=5`)で構築した従来の `iostream` ライブラリです。既存のソースで従来の `iostream` を使用している場合には、`libiostream` を使用すれば、ソースを修正しなくてもこれらのソースを標準モード (`-compat=5`) でコンパイルできます。このライブラリを使用するには、`-library=iostream` を使用します。

注 – 標準ライブラリのほとんどの部分は、標準 iostream を使用することに依存しています。従来の iostream を同一のプログラム内で使用すると、問題が発生する可能性があります。

- `libC`:これは互換モード (`-compat=4`) で必要なライブラリです。このライブラリには C++ 実行時サポートだけでなく従来の iostream も含まれています。
- `libcomplex`:このライブラリは、互換モード (`-compat=4`) で複素数の演算を行うときに必要です。標準モードの場合は、`libCstd` の複素数演算の機能が使用されます。
- `libstlport`:これは、C++ 標準ライブラリの STLport 実装です。このライブラリを使用するには、デフォルトの `libCstd` の代わりにオプション `-library=stlport4` を指定します。ただし、`libstlport` と `libCstd` の両方を同一プログラム内で使用することはできません。インポートしたライブラリを含み、すべてをどちらか一方のライブラリだけを使ってコンパイルしリンクしなければなりません。
- `librwtool` (`Tools.h++`):Tools.h++は、RogueWave の C++ 基礎クラスライブラリです。このリリースには、このライブラリのバージョン 7 が入っています。このライブラリには、従来の iostream 形式 (`-library=rwtools7`) と標準 iostream 形式 (`-library=rwtools7_std`) があります。このライブラリの詳細は、次のオンラインマニュアルを参照してください。
  - **Tools.h++** ユーザーズガイド (バージョン 7)
  - **Tools.h++** クラスライブラリ・リファレンスマニュアル (バージョン 7)

このマニュアルにアクセスするには、Web ブラウザで次のアドレスにアクセスしてください。

```
file:/opt/SUNWspro/docs/index.html
```

コンパイラソフトウェアが `/opt` ディレクトリにインストールされていない場合は、システム上でこのディレクトリに相当するパスをシステム管理者に問い合わせてください。

- libgc:このライブラリは、展開モードまたはガーベージコレクションモードで使用します。libgc ライブラリにリンクするだけで、プログラムのメモリーリークを自動的および永久的に修正することができます。プログラムを libgc ライブラリとリンクする場合は、free や delete を呼び出さずに、それ以外は通常どおりにプログラムを記述することができます。

詳細については、gcFixPrematureFrees(3) および gcInitialize(3) のマニュアルページを参照してください。

- libdemangle:このライブラリは、C++ 符号化名を復号化するときに使用します。

C++ ライブラリのマニュアルページへのアクセス

この節で説明しているライブラリに関するマニュアルページは次の場所にあります。

- /opt/SUNWspro/man/man1
- /opt/SUNWspro/man/man3
- /opt/SUNWspro/man/man3c++
- /opt/SUNWspro/man/man3cc4

注 – コンパイラソフトウェアが /opt ディレクトリにインストールされていない場合は、システム上でこのディレクトリに相当するパスをシステム管理者に問い合わせてください。

これらのマニュアルページにアクセスするには、MANPATH に /opt/SUNWspro/man (またはシステム上でこのディレクトリに相当するパス) が含まれていることを確認してください。MANPATH の設定方法については、このマニュアルの先頭にある「はじめに」のxxxii ページの「コンパイラコレクションのマニュアルへのアクセス」を参照してください。

C++ ライブラリのマニュアルページにアクセスするには次のとおり入力してください。

```
example% man library-name
```

C++ ライブラリのバージョン 4.2 のマニュアルページにアクセスするには次のコマンドを入力してください。

```
example% man -s 3CC4 library-name
```

次のアドレスに Web ブラウザでアクセスしてマニュアルページにアクセスすることもできます。

```
file:/opt/SUNWspro/docs/index.html
```

デフォルトの C++ ライブラリ

これらのライブラリには、CC ドライバによってデフォルトでリンクされるものと、明示的にリンクしなければならないものがあります。標準モードでは、次のライブラリが CC ドライバによってデフォルトでリンクされます。

```
-lCstd -lCrun -lm -lw -lc
```

互換モード (`-compat`) では、次のライブラリがデフォルトでリンクされます。

```
-lC -lm -lw -lc
```

詳細については、294 ページの「`-library=`l[,l...]」を参照してください。

関連するライブラリオプション

CC ドライバには、ライブラリを使用するためのオプションがいくつかあります。

- リンクするライブラリを指定するには、`-l` オプションを使用します。
- ライブラリを検索するディレクトリを指定するには、`-L` オプションを使用します。
- マルチスレッド化コードをコンパイルしてリンクするには、`-mt` オプションを使用します。
- 区間演算ライブラリをリンクするには、`-xia` オプションを使用します。

- Fortran 実行時ライブラリをリンクするには、-xlang オプションを使用します。
- Sun C++ コンパイラに添付された次のライブラリを指定するには、-library オプションを使用します。
  - libCrun
  - libCstd
  - libiostream
  - libC
  - libcomplex
  - libstlport, libstlport_dbg
  - librwtool, librwtool_dbg
  - libgc

注 – librwtool の従来の iostream 形式を使用するには、-library=rwtools7 オプションを使用します。librwtool の標準 iostream 形式を使用するには、-library=rwtools7_std オプションを使用します。

-library オプションと -staticlib オプションの両方に指定されたライブラリは静的にリンクされます。次にオプションの例をいくつか示します。

- 次のコマンドでは Tools.h++ バージョン 7 の従来の iostream 形式と libiostream ライブラリが動的にリンクされます。

```
example% CC test.cc -library=rwtools7,iostream
```

- 次のコマンドでは libgc ライブラリが静的にリンクされます。

```
example% CC test.cc -library=gc -staticlib=gc
```

- 次のコマンドでは test.cc が互換モードでコンパイルされ、libC が静的にリンクされます。互換モードでは -libC がデフォルトでリンクされるので、このライブラリを -library オプションで指定する必要はありません。

```
example% CC test.cc -compat=4 -staticlib=libC
```

■ 次のコマンドでは ライブラリ libCrun および libCstd がリンク対象から除外されます。指定しない場合は、これらのライブラリは自動的にリンクされます。

```
example% CC test.cc -library=no%Crun,no%Cstd
```

デフォルトでは、CC は、指定されたコマンド行オプションに従ってさまざなシステムライブラリをリンクします。-xnolib (または -nolib) を指定した場合、CC は、-l オプションを使用してコマンド行で明示的に指定したライブラリだけをリンクします (-xnolib または -nolib を使用した場合、-library オプションが存在していても無視されます)。

-R オプションは、動的ライブラリの検索パスを実行可能ファイルに組み込むときに使用します。実行時リンカーは、実行時にこれらのパスを使ってアプリケーションに必要な共有ライブラリを探します。CC ドライバは、デフォルトで -R/opt/SUNWspro/lib を ld に渡します (コンパイラが標準の場所にインストールされている場合)。共有ライブラリのデフォルトパスが実行可能ファイルに組み込まれないようにするには、-norunpath を使用します。

クラスライブラリの使用

一般に、クラスライブラリを使用するには 2 つの手順が必要です。

1. ソースコードに適切なヘッダーをインクルードする。
2. プログラムをオブジェクトライブラリとリンクする。

iostream ライブラリ

C++ コンパイラには、2 通りの iostream が実装されています。

■ **従来の iostream:** この用語は、C++ 4.0、4.0.1、4.1、4.2 コンパイラに添付された iostream ライブラリ、およびそれ以前に cfront ベースの 3.0.1 コンパイラに添付された iostream ライブラリを指します。このライブラリの標準はありませんが、既存のコードの多くがこれを使用しています。このライブラリは、互換モードの libC の一部であり、標準モードの libiostream にもあります。

- **標準の iostream** :これは C++ 標準ライブラリ `libCstd` に含まれていて、標準モードだけで使用されます。これは、バイナリレベルでもソースレベルでも「従来の iostream」とは互換性がありません。

すでに C++ のソースがある場合、そのコードは従来の iostream を使用しており、次の例のような形式になっていると思われます。

```
// file prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

次のコマンドは、互換性モードで `prog1.cc` をコンパイル、リンクして、`prog1` という実行可能なプログラムを生成します。従来の iostream ライブラリは、互換性モードのときにデフォルトでリンクされる `libC` ライブラリに含まれています。

```
example% CC -compat prog1.cc -o prog1
```

次の例では、標準の iostream が使用されています。

```
// file prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

次のコマンドは、`prog2.cc` をコンパイル、リンクして、`prog2` という実行可能なプログラムを生成します。コンパイルは標準モードで行われ、このモードでは、標準の iostream ライブラリを含む `libCstd` がデフォルトでリンクされます。

```
example% CC prog2.cc -o prog2
```

`libCstd` についての詳細は、第 13 章を参照してください。libiostream の詳細は、第 14 章を参照してください。

コンパイルモードの詳しい説明については、『C++ 移行ガイド』を参照してください。

complex ライブラリ

標準ライブラリには、C++ 4.2 コンパイラに付属していた complex ライブラリに似た、テンプレート化された complex ライブラリがあります。標準モードでコンパイルする場合は、<complex.h> ではなく、<complex> を使用する必要があります。互換性モードで <complex> を使用することはできません。

互換性モードでは、リンク時に complex ライブラリを明示的に指定しなければなりません。標準モードでは、complex ライブラリは libCstd に含まれており、デフォルトでリンクされます。

標準モード用の complex.h ヘッダーはありません。C++ 4.2 では、「complex」 はクラス名ですが、標準 C++ では「complex」はテンプレート名です。したがって、旧式のコードを変更せずに動作できるようにする typedef を使用することはできません。このため、複素数を使用する、4.2 用のコードで標準ライブラリを使用するには、多少の編集が必要になります。たとえば、次のコードは 4.2 用に作成されたものであり、互換性モードでコンパイルされます。

```
// file ex1.cc (compatibility mode)
#include <iostream.h>
#include <complex.h>

int main()
{
    complex x(3,3), y(4,4);
    complex z = x * y;
    cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
}
```

次の例では、ex1.cc を互換モードでコンパイル、リンクし、生成されたプログラムを実行しています。

```
example% CC -compat ex1.cc -library=complex
example% a.out
x=(3, 3), y=(4, 4), z=(0, 24)
```

次は、標準モードでコンパイルされるように ex2.cc と書き直された ex1.cc です。

```
// file ex2.cc (ex1.cc rewritten for standard mode)
#include <iostream>
#include <complex>
using std::complex;

int main()
{
     complex<double> x(3,3), y(4,4);
     complex<double> z = x * y;
     std::cout << "x=" << x << ", y=" << y << ", z=" << z <<
       std::endl;
}
```

次の例では、書き直された ex2.cc をコンパイル、リンクして、生成されたプログラムを実行しています。

```
% CC ex2.cc
% a.out
x=(3,3), y=(4,4), z=(0,24)
```

複素数演算ライブラリの使用方法についての詳細は、第 15 章を参照してください。

C++ライブラリのリンク

次の表は、C++ ライブラリにリンクするためのコンパイラオプションをまとめています。詳細については、294 ページの「-library=l[,l...]」を参照してください。

表 12-2 C++ ライブラリにリンクするためのコンパイラオプション

ライブラリ	コンパイルモード	オプション
従来の iostream	-compat=4	不要
	-compat=5	-library=iostream
complex	-compat=4	-library=complex
	-compat=5	不要
Tools.h++ version 7	-compat=4	-library=rwtools7
	-compat=5	-library=rwtools7,iostream
		-library=rwtools7_std
Tools.h++ version 7 debug	-compat=4	-library=rwtools7_dbg
	-compat=5	-library=rwtools7_dbg,iostream
		-library=rwtools7_std_dbg
ガベージコレクション	-compat=4	-library=gc
	-compat=5	-library=gc
STLport バージョン 4	-compat=5	-library=stlport4
STLport バージョン 4 デバッグ	-compat=5	-library=stlport4_dbg

標準ライブラリの静的リンク

デフォルト時、CC ドライバは、デフォルトライブラリの-l*lib* オプションをリンカーに渡すことによって、libc と libm を含むいくつかのライブラリの共有バージョンでリンクします(互換性モードと標準モードにおけるデフォルトライブラリのリストについては、160 ページの「デフォルトの C++ ライブラリ」 を参照してください)。

このようにデフォルトのライブラリを静的にリンクする場合、-library オプションと -staticlib オプションを一緒に使用すれば、C++ ライブラリを静的にリンクできます。この方法は、以前説明した方法よりもかなり簡単です。構築の例を次に示します。

```
example% CC test.c -staticlib=Crun
```

この例では、-library オプションが明示的にコマンドに指定されていません。標準モード (デフォルトのモード) では、-library のデフォルトの設定が Cstd,Crun であるため、-library オプションを明示的に指定する必要はありません。

あるいは、-xnolib コンパイラオプションも使用できます。-xnolib オプションを指定すると、ドライバは自動的には -l オプションを ld に渡しません。-l オプションは、自分で渡す必要があります。次の例は、Solaris 2.6、Solaris 7、Solaris 8 のいずれかのオペレーティング環境で libCrun と静的に、libw、libm、libc と動的にリンクする方法を示します。

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun \
-Bdynamic -lm -lw -lcx -lc
```

-l オプションの順序は重要です。-lc の前に -lCstd、-lCrun、-lm、-lw、-lcx オプションがあることに注意してください。

注 – IA プラットフォームでは、-lcx オプションはありません。

他のライブラリにリンクする CC オプションもあります。そうしたライブラリへのリンクも -xnolib によって行われないように設定できます。たとえば、-mt オプションを指定すると、CC ドライバは、-lthread を ld に渡します。これに対し、-mt と -xnolib の両方を使用すると、CC ドライバは ld に -lthread を渡しません。詳細については、372 ページの「-xnolib」を参照してください。ld については、Solaris に関するマニュアル『リンカーとライブラリ』を参照してください。

共有ライブラリの使用

C++ コンパイラには、次の共有ライブラリが含まれています。

- libCrun.so
- libC.so
- libcomplex.so
- libstlport.so
- librwtool.so
- libgc.so
- libgc_dbg.so
- libCstd.so
- libiostream.so

プログラムにリンクされた各共有オブジェクトは、生成される実行可能ファイル (a.out ファイル) に記録されます。この情報は、実行時に ld.so が使用して動的リンク編集を行います。ライブラリコードをアドレス空間に実際に組み込むのは後になるため、共有ライブラリを使用するプログラムの実行時の動作は、環境の変化 (つまり、ライブラリを別のディレクトリに移動すること) に影響を受けます。たとえば、プログラムが /opt/SUNWspro/lib の libcomplex.so.5 とリンクされている場合、後で libcomplex.so.5 ライブラリを /opt2/SUNWspro/lib に移動すると、このバイナリコードを実行したときに次のメッセージが表示されます。

```
ld.so: libcomplex.so.5: not found
```

ただし、環境変数LD_LIBRARY_PATH に新しいライブラリの ディレクトリを設定すれば、古いバイナリコードを再コンパイルせずに実行できます。

C シェルでは次のように入力します。

```
example% setenv LD_LIBRARY_PATH \
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
```

Bourne シェルでは次のように入力します。

```
example$ LD_LIBRARY_PATH=\
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
example$ export LD_LIBRARY_PATH
```

注 – *release* は、コンパイラソフトウェアのそれぞれのリリースによって異なります。

LD_BINARY_PATH には、ディレクトリのリストが含まれています。ディレクトリは通常コロンで区切られています。C++ プログラムを実行すると、動的ローダーがデフォルトディレクトリより前に LD_BINARY_PATH のディレクトリを検索します。

実行可能ファイルにどのライブラリが動的にリンクされるのかを知るには、ldd コマンドを使用します。

```
example% ldd a.out
```

共有ライブラリを移動することはめったにないので、この手順が必要になることはほとんどありません。

注 – 共有ライブラリを dlopen で開く場合は、RTLD_GLOBAL を使用しないと例外が機能しません。

共有ライブラリの詳しい使い方については、『リンカーとライブラリ』を参照してください。

C++ 標準ライブラリの置き換え

ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。基本的な操作としては、コンパイラに添付されている標準のヘッダーとライブラリを無効にして、新しいヘッダーファイルとライブラリが格納されているディレクトリとライブラリ自身の名前を指定します。

コンパイラは、標準ライブラリの STLport 実装をサポートします。詳細については、193 ページの「STLport」を参照してください。

置き換え可能な対象

ほとんどの標準ライブラリおよびそれに関連するヘッダーは置き換え可能です。たとえば libCstd ライブラリを別のものに置き換える場合は、次の関連するヘッダーも置き換える必要があります。

```
<algorithm> <bitset> <complex> <deque> <fstream> <functional>
<iomanip> <ios> <iosfwd> <iostream> <istream> <iterator> <limits>
<list> <locale> <map> <memory> <numeric> <ostream> <queue> <set>
<sstream> <stack> <stdexcept> <streambuf> <string> <strstream>
<utility> <valarray> <vector>
```

ライブラリの置き換え可能な部分は、いわゆる「STL」と呼ばれているもの、文字列クラス、iostream クラス、およびそれらの補助クラスです。このようなクラスとヘッダーは相互に依存しているため、それらの一部を置き換えるだけでは通常は機能しません。一部を変更する場合でも、すべてのヘッダーと libCstd のすべてを置き換える必要があります。

置き換え不可能な対象

標準ヘッダー <exception>、<new>、および <typeinfo> は、コンパイラ自身と libCrun に密接に関連しているため、これらを置き換えることは安全ではありません。ライブラリ libCrun は、コンパイラが依存している多くの「補助」関数が含まれているため置き換えることはできません。

C から派生した 17 個の標準ヘッダー (<stdlib.h>、<stdio.h>、<string.h> など) は、Solaris オペレーティング環境と基本 Solaris 実行時ライブラリ libc に密接に関連しているため、これらを置き換えることは安全ではありません。これらのヘッダーの C++ 版 (<cstdlib>、<cstdio>、<cstring> など) は基本の C 版のヘッダーに密接に関連しているため、これらを置き換えることは安全ではありません。

代替ライブラリのインストール

代替ライブラリをインストールするには、まず、代替ヘッダーの位置と libCstd の代わりに使用するライブラリを決定する必要があります。理解しやすくするために、ここでは、ヘッダーを /opt/mycstd/include にインストールし、ライブラリを /opt/mycstd/lib にインストールすると仮定します。ライブラリの名前は libmyCstd.a であると仮定します。なお、ライブラリの名前を lib で始めると後々便利です。

代替ライブラリの使用

コンパイルごとに -I オプションを指定して、ヘッダーがインストールされている位置を指示します。さらに、-library=no%Cstd オプションを指定して、コンパイラ独自のバージョンの libCstd ヘッダーが検出されないようにします。構築の例を次に示します。

```
example% CC -I/opt/mycstd/include -library=no%Cstd ... (compile)
```

-library=no%Cstd オプションを指定しているため、コンパイル中、コンパイラ独自のバージョンのヘッダーがインストールされているディレクトリは検索されません。

プログラムまたはライブラリのリンクごとに -library=no%Cstd オプションを指定して、コンパイラ独自の libCstd が検出されないようにします。さらに、-L オプションを指定して、代替ライブラリがインストールされているディレクトリを指示します。さらに、-l オプションを指定して、代替ライブラリを指定します。例:

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd ... (link)
```

あるいは、-L や -l オプションを使用せずに、ライブラリの絶対パス名を直接指定することもできます。For example:

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a ... (link)
```

-library=no%Cstd オプションを指定しているため、リンク中、コンパイラ独自のバージョンの libCstd はリンクされません。

標準ヘッダーの実装

C には、<stdio.h>、<string.h>、<stdlib.h> などの 17 個の標準ヘッダーがあります。これらのヘッダーは Solaris オペレーティング環境に標準で付属しており、/user/include に置かれています。C++ にも同様のヘッダーがありますが、さまざまな宣言の名前が大域の名前空間と std 名前空間の両方に存在するという条件が付加されています。バージョン 8 より前のリリースの Solaris オペレーティング環境の C++ コンパイラでは、/usr/include ディレクトリにあるヘッダーはそのまま残して、独自のバージョンのヘッダーを別に用意しています。

また、C++ には、C 標準ヘッダー (<cstdio>、<cstring>、<cstdlib> など) のそれぞれについても専用のバージョンがあります。C++ 版の C 標準ヘッダーでは、宣言名は std 名前空間にのみ存在します。C++ には、32 個の独自の標準ヘッダー (<string>、<utility>、<iostream> など) も追加されています。

標準ヘッダーの実装で、C++ ソースコード内の名前がインクルードするテキストファイル名として使用されているとしましょう。たとえば、標準ヘッダーの <string> (または <string.h>) が、あるディレクトリにある string (または string.h) というファイルを参照するものとします。この実装には、次の欠点があります。

- ファイル名に接尾辞がない場合、ヘッダーファイルだけ検索したり、ヘッダーファイル用のmakefile 規則を作成したりできない。
- コンパイラのコマンド行に -I/usr/include を指定すると、コンパイラ専用の include ディレクトリの前に /usr/include が検索されるため、Solaris 2.6 および Solaris 7 オペレーティング環境の正しいバージョンの標準 C ヘッダーが検出されない。
- string というディレクトリまたは実行可能プログラムがあると、そのディレクトリまたはプログラムが標準ヘッダーファイルの代わりに検出される可能性がある。
- Solaris 8 オペレーティング環境より前のリリースの Solaris オペレーティング環境では .KEEP_STATE が有効なときの makefile のデフォルトの相互依存関係により、標準ヘッダーが実行可能プログラムに置き換えられる可能性がある(デフォルトの場合、接尾辞がないファイルは構築対象プログラムとみなされる)。

こうした問題を解決するため、コンパイラの include ディレクトリには、ヘッダーと同じ名前を持つファイルと、一意の接尾辞 SUNWCCh を持つ、そのファイルへのシンボリックリンクが含まれています (SUNW はコンパイラに関係するあらゆるパッケージに対する接頭辞、CC は C++ コンパイラの意味、.h はヘッダーファイルの通常の接尾辞)。つまり <string> と指定された場合、コンパイラは <string.SUNWCCh> と書き換え、その名前を検索します。接尾辞付きの名前は、コンパイラ専用の include ディレクトリにだけ存在します。このようにして見つけられたファイルがシンボリックリンクの場合 (通常はそうである)、コンパイラは、エラーメッセージやデバッガの参照でそのリンクを 1 回だけ間接参照し、その参照結果 (この場合は string) をファイル名として使用します。ファイルの依存関係情報を送るときは、接尾辞付きの名前の方が使用されます。

この名前の書き換えは、2 つのバージョンがある 17 個の標準 C ヘッダーと 32 個の標準 C++ヘッダーのいずれかを、パスを指定せずに山括弧 < > で囲んで指定した場合にだけ行われます。山括弧の代わりに引用符が使用されるか、パスが指定されるか、他のヘッダーが指定された場合、名前の書き換えは行われません。

次の表は、よくある書き換え例をまとめています。

表 12-3 ヘッダー検索の例

ソースコード	コンパイラによる検索	注釈
`<string>`	`string.SUNWCCh`	C++ の文字列テンプレート
`<cstring>`	`cstring.SUNWCCh`	C の `string.h` の C++ 版
`<string.h>`	`string.h.SUNWCCh`	C `string.h`
`<fcntl.h>`	`fcntl.h`	標準 C および C++ヘッダー以外
`"string"`	`string`	山括弧ではなく、二重引用符
`<../string>`	`../string`	パス指定がある場合

コンパイラが *header*`.SUNWCCh` (*header* はヘッダー名) を見つけることができなかった場合、コンパイラは、`#include` 指令で指定された名前で検索をやり直します。たとえば、`#include <string>` という指令を指定した場合、コンパイラは `string.SUNWCCh` という名前のファイルを見つけようとします。この検索が失敗した場合、コンパイラは `string` という名前のファイルを探します。

標準 C++ ヘッダーの置き換え

171 ページの「標準ヘッダーの実装」で説明している検索アルゴリズムのため、170 ページの「代替ライブラリのインストール」で説明している `SUNWCCh` 版の代替ヘッダーを指定する必要はありません。しかし、これまでに説明したいくつかの問題が発生する可能性もあります。その場合、推奨される解決方法は、接尾辞が付いていないヘッダーごとに、接尾辞 `.SUNWCCh` を持つファイルに対してシンボリックリンクを作成することです。つまり、ファイルが `utility` の場合、次のコマンドを実行します。

```
example% ln -s utility utility.SUNWCCh
```

utility.SUNWCCh というファイルを探すとき、コンパイラは 1 回目の検索でこのファイルを見つけます。そのため、utility という名前の他のファイルやディレクトリを誤って検出してしまうことはありません。

標準 C ヘッダーの置き換え

標準 C ヘッダーの置き換えはサポートされていません。それでもなお、独自のバージョンの標準ヘッダーを使用したい場合、推奨される手順は次のとおりです。

- すべての代替ヘッダーを 1 つのディレクトリに置きます。
- そのディレクトリ内にある代替ヘッダーごとに *header*.SUNWCCh (*header* はヘッダー名) へのシンボリックリンクを作成します。
- コンパイラを呼び出すごとに -I 指令を指定して、代替ヘッダーが置かれているディレクトリが検索されるようにします。

たとえば、<stdio.h> と <cstdio> の代替ヘッダーとして stdio.h と cstdio を使用したいとします。stdio.h と cstdio をディレクトリ /myproject/myhdr に置きます。このディレクトリ内で、次のコマンドを実行します。

```
example% ln -s stdio.h stdio.h.SUNWCCh
example% ln -s cstdio cstdio.SUNWCCh
```

コンパイルのたびに、オプション -I/myproject/mydir を使用します。

警告:

- C ヘッダーを置き換える場合は、対になっているもう一方のヘッダーを置き換える必要があります。たとえば、<time.h> を置き換えるときは、<ctime> も置き換える必要があります。
- 代替ヘッダーは、置き換える前のヘッダーと同じ効果を持っている必要があります。これは、さまざまな実行時ライブラリ (libCrun、libC、libCstd、libc、および librwtool) が標準ヘッダーの定義を使用して構築されているためです。同じ効果を持っていない場合、作成したプログラムはほとんどの場合、正しく動作しません。

第13章

C++ 標準ライブラリの使用

デフォルトモード (標準モード) のコンパイルでは、コンパイラは C++ 標準で指定されている完全なライブラリにアクセスします。このライブラリには、非公式に「標準テンプレートライブラリ」 (STL) と呼ばれているものに加えて、次の要素が含まれています。

- 文字列クラス
- 数値クラス
- 標準のストリーム入出力クラス
- 基本的なメモリー割り当て
- 例外クラス
- 実行時の型識別 (RTTI)

STL は公式なものではありませんが、一般的にはコンテナ、反復子、アルゴリズムから構成されます。標準ライブラリのヘッダーのうち、次のものを STL の構成要素と見なすことができます。

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

C++ 標準ライブラリ (libCstd) は、RogueWave の標準 C++ ライブラリに基づいています。このライブラリはデフォルトモード (-compat=5) だけで使用することができます。-compat=4 オプションを使用した場合はサポートされません。

また、C++ コンパイラで、STLport の標準ライブラリのバージョン 4.5.3 がサポートされました。libCstd がデフォルトのライブラリですが、代わりに STLport の製品を使用できるようになりました。詳細については、193 ページの「STLport」を参照してください。

コンパイラに付属している C++ 標準ライブラリの代わりに、独自の C++ 標準ライブラリを使用することもできます。その場合は、-library=no%Cstd オプションを使用します。ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。詳細は、169 ページの「C++ 標準ライブラリの置き換え」を参照してください。

標準ライブラリの詳細については、『標準 C++ ライブラリ・ユーザーズガイド』と『Standard C++ Library Class Reference』(英語版) を参照してください。これらの文書へのアクセス方法については、本書の冒頭にある 「はじめに」 のxxxii ページの「コンパイラコレクションのマニュアルへのアクセス」を参照してください。また、C++ 標準ライブラリについての参考書については、「はじめに」 の xxxvi ページの「市販の書籍」を参照してください。

C++ 標準ライブラリのヘッダーファイル

標準ライブラリのヘッダーとその概要を表 13-1に示します。

表 13-1 C++ 標準ライブラリのヘッダーファイル

ヘッダーファイル	内容
<algorithm>	コンテナ操作のための標準アルゴリズム
<bitset>	固定長のビットシーケンス
<complex>	複素数を表す数値型
<deque>	先頭と末尾の両方で挿入と削除が可能なシーケンス
<exception>	事前定義済み例外クラス
<fstream>	ファイルとのストリーム入出力
<functional>	関数オブジェクト

表 13-1 C++ 標準ライブラリのヘッダーファイル (続き)

ヘッダーファイル	内容
<iomanip>	iostream manipulators
<ios>	iostream base classes
<iosfwd>	iostream クラスの先行宣言
<iostream>	基本的なストリーム入出力機能
<istream>	入力ストリーム
<iterator>	シーケンスの内容にくまなくアクセスするためのクラス
<limits>	数値型の属性
<list>	順序付きシーケンス
<locale>	国際化のサポート
<map>	キーと値を対にして使用する連想コンテナ
<memory>	特殊なメモリーアロケータ
<new>	基本的なメモリー割り当てと解放
<numeric>	汎用の数値演算
<ostream>	出力ストリーム
<queue>	先頭への挿入と末尾からの削除が可能なシーケンス
<set>	一意キーを使用する連想コンテナ
<sstream>	メモリー上の文字列との入出力ストリーム
<stack>	先頭への挿入と先頭からの削除が可能なシーケンス
<stdexcept>	追加標準例外クラス
<streambuf>	iostream 用のバッファークラス
<string>	文字シーケンス
<typeinfo>	実行時の型識別
<utility>	比較演算子
<valarray>	数値プログラミング用の値配列
<vector>	ランダムアクセスが可能なシーケンス

C++ 標準ライブラリのマニュアルページ

標準ライブラリの個々の構成要素のマニュアルページを 表 13-2 に示します。

表 13-2 C++ 標準ライブラリのマニュアルページ

マニュアルページ	概要
Algorithms	コンテナとシーケンスに各種処理を行うための汎用アルゴリズム
Associative_Containers	特定の順序で並んだコンテナ
Bidirectional_Iterators	読み書きの両方が可能で、順方向、逆方向にコンテナをたどることができる反復子
Containers	標準テンプレートライブラリ (STL) コレクション
Forward_Iterators	読み書きの両方が可能な順方向反復子
Function_Objects	operator() が定義済みのオブジェクト
Heap_Operations	make_heap、pop_heap、push_heap、sort_heap を参照
Input_Iterators	読み取り専用の順方向反復子
Insert_Iterators	反復子がコンテナ内の要素を上書きせずにコンテナに挿入することを可能にする、反復子アダプタ
Iterators	コレクションをたどったり、変更したりするためのポインタ汎用化機能
Negators	述語関数オブジェクトの意味を逆にするための関数アダプタと関数オブジェクト
Operators	C++ 標準テンプレートライブラリ出力用の演算子
Output_Iterators	書き込み専用の順方向反復子
Predicates	ブール値 (真偽) または整数値を返す関数または関数オブジェクト
Random_Access_Iterators	コンテナの読み取りと書き込みをして、コンテナにランダムアクセスすることを可能にする反復子
Sequences	一群のシーケンスをまとめたコンテナ

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
Stream_Iterators	汎用アルゴリズムをストリームに直接に使用することを可能にする、ostream と istream 用の反復子機能を含む
__distance_type	反復子が使用する距離のタイプを決定する (廃止予定)
__iterator_category	反復子が属するカテゴリを決定する (廃止予定)
__reverse_bi_iterator	コレクションを逆方向にたどる反復子
accumulate	1 つの範囲内のすべての要素の累積値を求める
adjacent_difference	1 つの範囲内の隣り合う 2 つの要素の差のシーケンスを出力する
adjacent_find	シーケンスから、等しい値を持つ最初の 2 つの要素を検出する
advance	特定の距離で、順方向または逆方向 (使用可能な場合) に反復子を移動する
allocator	標準ライブラリコンテナ内の記憶管理用のデフォルトの割り当てオブジェクト
auto_ptr	単純でスマートなポインタクラス
back_insert_iterator	コレクションの末尾への項目の挿入に使用する挿入反復子
back_inserter	コレクションの末尾への項目の挿入に使用する挿入反復子
basic_filebuf	入力または出力シーケンスをファイルに関連付けるクラス
basic_fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
basic_ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りサポートする
basic_ios	すべてのストリームが必要とする共通の関数を含む基底クラス

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
basic_iostream	ストリームバッファーが制御する文字シーケンスの書式設定と解釈をサポートする
basic_istream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
basic_istringstream	メモリー上の配列からの basic_string<chartT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
basic_ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
basic_ostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
basic_ostringstream	basic_string<chart,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
basic_streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
basic_string	文字に似た要素シーケンスを処理するためのテンプレート化されたクラス
basic_stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
basic_stringstream	メモリー上の配列に対する basic_string<chartT,traits,Allocator> クラスのオブジェクトの書き込みと読み取りをサポートする
binary_function	2 項関数オブジェクトを作成するための基底クラス
binary_negate	2 項判定子の結果の補数を返す関数オブジェクト
binary_search	コンテナ上の値について 2 等分検索を行う
bind1st	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ
bind2nd	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
binder1st	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ
binder2nd	関数オブジェクトに値を結合するためのテンプレート化されたユーティリティ
bitset	固定長のビットシーケンスを格納、操作するためのテンプレートクラスと関数
cerr	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたバッファリングなしストリームバッファーに対する出力を制御する
char_traits	basic_string コンテナと iostream クラス用の型と演算を持つ特性 (traits) クラス
cin	<cstdio> で宣言されたオブジェクトの stdin に関連付けられたストリームバッファーからの入力を制御する
clog	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
codecvt	コード変換ファセット
codecvt_byname	指定ロケールに基づいたコードセット変換分類機能を含むファセット
collate	文字列照合、比較、ハッシュファセット
collate_byname	文字列照合、比較、ハッシュファセット
compare	真または偽を返す 2 項関数または関数オブジェクト
complex	C++ 複素数ライブラリ
copy	ある範囲の要素をコピーする
copy_backward	ある範囲の要素をコピーする
count	指定条件を満たすコンテナ内の要素の個数をカウントする
count_if	指定条件を満たすコンテナ内の要素の個数をカウントする

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
cout	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
ctype	文字分類機能を取り込むファセット
ctype_byname	指定ロケールに基づいた文字分類機能を含むファセット
deque	ランダムアクセス反復子と、先頭および末尾の両方での効率的な挿入と削除をサポートするシーケンス
distance	2 つの反復子間の距離を求める
divides	1 つ目の引数を 2 つ目の引数で除算した結果を返す
equal	2 つのある範囲が等しいかどうか比較する
equal_range	並べ替えの順序を崩さずに値を挿入できる最大の二次範囲をコレクションから検出する
equal_to	1 つ目と 2 つ目の引数が等しい場合に真を返す 2 項関数オブジェクト
exception	倫理エラーと実行時エラーをサポートするクラス
facets	複数種類のロケール機能をカプセル化するために使用するクラス群
filebuf	入力または出力シーケンスをファイルに関連付けるクラス
fill	指定された値である範囲を初期化する
fill_n	指定された値である範囲を初期化する
find	シーケンスから値に一致するものを検出する
find_end	シーケンスからサブシーケンスに最後に一致するものを検出する
find_first_of	シーケンスから、別のシーケンスの任意の値に一致するものを検出する
find_if	シーケンスから指定された判定子を満たす値に一致するものを検出する
for_each	ある範囲のすべての要素に関数を適用する
fpos	iostream クラスの位置情報を保持する

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
front_insert_iterator	コレクションの先頭に項目を挿入するための挿入反復子
front_inserter	コレクションの先頭に項目を挿入するための挿入反復子
fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
generate	値生成クラスによって生成された値でコンテナを初期化する
generate_n	値生成クラスによって生成された値でコンテナを初期化する
get_temporary_buffer	メモリーを処理するためのポインタベースのプリミティブ
greater	1 つ目の引数が 2 つ目の引数より大きい場合に真を返す 2 項関数オブジェクト
greater_equal	1 つ目の引数が 2 つ目の引数より大きいか等しい場合に真を返す 2 項関数オブジェクト
gslice	配列から汎用化されたスライスを表現するために使用される数値配列クラス
gslice_array	valarray から BLAS に似たスライスを表現するために使用される数値配列クラス
has_facet	ロケールに指定ファセットがあるかどうかを判定するための関数テンプレート
ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りをサポートする
includes	ソートされたシーケンスに対する基本演算セット
indirect_array	valarray から選択された要素の表現に使用される数値配列クラス
inner_product	2 つの範囲 A および B の内積 (A × B) を求める
inplace_merge	ソートされた 2 つのシーケンスを 1 つにマージする

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
insert_iterator	コレクションを上書きせずにコレクションに項目を挿入するときに使用する挿入反復子
inserter	コレクションを上書きせずにコレクションに項目を挿入するときに使用する挿入反復子
ios	すべてのストリームが必要とする共通の関数を含む基底クラス
ios_base	メンバーの型を定義して、そのメンバーから継承するクラスのデータを保持する
iosfwd	入出力ライブラリテンプレートクラスを宣言し、そのクラスを wide および tiny 型文字専用にする
isalnum	文字が英字または数字のどちらであるかを判定する
isalpha	文字が英字であるかどうかを判定する
iscntrl	文字が制御文字であるかどうかを判定する
isdigit	文字が 10 進数であるかどうかを判定する
isgraph	文字が図形文字であるかどうかを判定する
islower	文字が英小文字であるかどうかを判定する
isprint	文字が印刷可能かどうかを判定する
ispunct	文字が区切り文字であるかどうかを判定する
isspace	文字が空白文字であるかどうかを判定する
istream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
istream_iterator	istream に対する反復子機能を持つストリーム反復子
istreambuf_iterator	作成元のストリームバッファーから連続する文字を読み取る
istringstream	メモリー上の配列からの basic_string<chartT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
istrstream	メモリー上の配列から文字を読み取る
isupper	文字が英大文字であるかどうかを判定する
isxdigit	文字が 16 進数であるかどうかを判定する

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
iter_swap	2 つの位置の値を交換する
iterator	基底反復子クラス
iterator_traits	反復子に関する基本的な情報を返す
less	1 つ目の引数が 2 つ目の引数より小さい場合に真を返す 2 項関数オブジェクト
less_equal	1 つ目の引数が 2 つ目の引数より小さいか、等しい場合に真を返す 2 項関数オブジェクト
lexicographical_compare	2 つの範囲を辞書式に比較する
limits	numeric_limits セクションを参照
list	双方向反復子をサポートするシーケンス
locale	多相性を持つ複数のファセットからなる地域対応化クラス
logical_and	1 つ目の 2 つ目の引数が等しい場合に真を返す場合に 2 項関数オブジェクト
logical_not	引数が偽の場合に真を返す単項関数オブジェクト
logical_or	引数のいずれかが真の場合に真を返す 2 項関数オブジェクト
lower_bound	ソートされたコンテナ内の最初に有効な要素位置を求める
make_heap	ヒープを作成する
map	一意のキーを使用してキー以外の値にアクセスする連想コンテナ
mask_array	valarray の選別ビューを提供する数値配列クラス
max	2 つの値の大きい方の値を検出して返す
max_element	1 つの範囲内の最大値を検出する
mem_fun	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun1	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun_ref	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
mem_fun_ref1	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
merge	ソートされた 2 つのシーケンスをマージして、3 つ目のシーケンスを作成する
messages	メッセージ伝達ファセット
messages_byname	メッセージ伝達ファセット
min	2 つの値の小さい方の値を検出して返す
min_element	1 つの範囲内の最小値を検出する
minus	1 つ目の引数から 2 つ目の引数を減算した結果を返す
mismatch	2 つのシーケンスの要素を比較して、互いに値が一致しない最初の 2 つの要素を返す
modulus	1 つ目の引数を 2 つ目の引数で除算することによって得られた余りを返す
money_get	入力に対する通貨書式設定ファセット
money_put	出力に対する通貨書式設定ファセット
moneypunct	通貨句読文字ファセット
moneypunct_byname	通貨句読文字ファセット
multimap	キーを使用してコンテナキーでない値にアクセスするための連想コンテナ
multiplies	1 つ目と 2 つ目の引数を乗算した結果を返す 2 項関数オブジェクト
multiset	格納済みのキー値に高速アクセスするための連想コンテナ
negate	引数の否定値を返す単項関数オブジェクト
next_permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
not1	単項述語関数オブジェクトの意味を逆にするための関数アダプタ
not2	単項述語関数オブジェクトの意味を逆にするための関数アダプタ

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
not_equal_to	1 つ目の引数が 2 つ目の引数と等しくない場合に真を返す 2 項関数オブジェクト
nth_element	コレクションを再編して、ソートで *n* 番目の要素より後になった全要素をその要素より前に、*n* 番目の要素より前の全要素をその要素より後ろにくるようにする
num_get	入力に対する書式設定ファセット
num_put	出力に対する書式設定ファセット
numeric_limits	スカラー型に関する情報を表すためのクラス
numpunct	数値句読文字ファセット
numpunct_byname	数値句読文字ファセット
ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
ostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
ostream_iterator	ostream と istream に反復子を使用可能にするストリーム反復子
ostreambuf_iterator	作成元のストリームバッファーに連続する文字を書き込む
ostringstream	basic_string<chart,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
ostrstream	メモリー上の配列に書き込みを行う
pair	異種の値の組み合わせ用テンプレート
partial_sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
partial_sort_copy	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
partial_sum	ある範囲の値の連続した部分小計を求める
partition	指定述語を満たす全エンティティを、満たさない全エンティティの前に書き込む

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
plus	1 つ目と 2 つ目の引数を加算した結果を返す 2 項関数オブジェクト
pointer_to_binary_function	binary_function の代わりとしてポインタを 2 項関数に適用する関数オブジェクト
pointer_to_unary_function	unary_function の代わりとしてポインタを関数に適用する関数オブジェクトクラス
pop_heap	ヒープの外に最大要素を移動する
prev_permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
priority_queue	優先順位付きの待ち行列のように振る舞うコンテナアダプタ
ptr_fun	関数の代わりとしてポインタを関数に適用するときに多重定義される関数
push_heap	ヒープに新しい要素を書き込む
queue	先入れ先出しの待ち行列のように振る舞うコンテナアダプタ
random_shuffle	コレクションの要素を無作為にシャッフルする
raw_storage_iterator	反復子ベースのアルゴリズムが初期化されていないメモリーに結果を書き込めるようにする
remove	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_copy	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_copy_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
replace	コレクション内の要素の値を置換する
replace_copy	コレクション内の要素の値を置換して、置換後のシーケンスを結果に移動する

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
replace_copy_if	コレクション内の要素の値を置換して、置換後のシーケンスを結果に移動する
replace_if	コレクション内の要素の値を置換する
return_temporary_buffer	メモリーを処理するためのポインタベースのプリミティブ
reverse	コレクション内の要素を逆順にする
reverse_copy	コレクション内の要素を逆順にしながら、その結果を新しいコレクションにコピーする
reverse_iterator	コレクションを逆方向にたどる反復子
rotate	先頭から中央直前の要素までのセグメントと中央から末尾までの要素のセグメントを交換する
rotate_copy	先頭から中央直前の要素までのセグメントと中央から末尾までの要素のセグメントを交換する
search	値シーケンスから、要素単位で指定範囲の値に等しいサブシーケンスを検出する
search_n	値シーケンスから、要素単位で指定範囲の値に等しいサブシーケンスを検出する
set	一意のキーを扱う連想コンテナ
set_difference	ソートされた差を作成する基本的な集合演算
set_intersection	ソートされた積集合を作成する基本的な集合演算
set_symmetric_difference	ソートされた対称差を作成する基本的な集合演算
set_union	ソートされた和集合を作成する基本的な集合演算
slice	配列の BLAS に似たスライスを表す数値配列クラス
slice_array	valarray の BLAS に似たスライスを表す数値配列クラス
smanip	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
smanip_fill	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
sort_heap	ヒープをソートされたコレクションに変換する
stable_partition	各グループ内の要素の相対的な順序を保持しながら、指定判定子を満たす全エンティティを満たさない全エンティティの前に書き込む
stable_sort	エンティティのコレクションをソートするためのテンプレート化されたアルゴリズム
stack	先入れ先出しのスタックのように振る舞うコンテナアダプタ
streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
string	basic_string<char、char_traits<char>、allocator<char>> 用の型定義
stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
stringstream	メモリー上の配列に対する basic_string<chartT、traits、Allocator> クラスのオブジェクトの書き込みおよび読み取りをサポートする
strstream	メモリー上の配列に対する読み取りと書き込みを行う
strstreambuf	入力または出力シーケンスを、要素が任意の値を格納する超小型の文字配列に関連付ける
swap	値を交換する
swap_ranges	ある位置の値の範囲を別の位置の値と交換する
time_get	入力に対する時刻書式設定ファセット
time_get_byname	指定ロケールに基づいた、入力に対する時刻書式設定ファセット
time_put	入力に対する時刻書式設定ファセット
time_put_byname	指定ロケールに基づいた、入力に対する時刻書式設定ファセット
tolower	文字を小文字に変換する

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
toupper	文字を大文字に変換する
transform	コレクション内の値の範囲に演算を適用し、結果を格納する
unary_function	単項関数オブジェクトを作成するための基底クラス
unary_negate	単項述語の結果の補数を返す関数オブジェクト
uninitialized_copy	構造構文を使用してある範囲の値を別の位置にコピーするアルゴリズム
uninitialized_fill	コレクション内の値の設定に構造構文アルゴリズムを使用するアルゴリズム
uninitialized_fill_n	コレクション内の値の設定に構造構文アルゴリズムを使用するアルゴリズム
unique	1 つの範囲の値から連続する重複値を削除し、得られた一意の値を結果に書き込む
unique_copy	1 つの範囲の値から連続する重複値を削除し、得られた一意の値を結果に書き込む
upper_bound	ソートされたコンテナ内の最後に有効な値位置を求める
use_facet	ファセットの取得に使用するテンプレート関数
valarray	数値演算用に最適化された配列クラス
vector	ランダムアクセス反復子をサポートするシーケンス
wcerr	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたバッファリングなしストリームバッファーに対する出力を制御する
wcin	<cstdio> で宣言されたオブジェクトの stdin に関連付けられたストリームバッファーからの 入力を制御する
wclog	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
wcout	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する

表 13-2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
wfilebuf	入力または出力シーケンスをファイルに関連付けるクラス
wfstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
wifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りをサポートする
wios	すべてのストリームが必要とする共通の関数を含む基底クラス
wistream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
wistringstream	メモリー上の配列からの basic_string<chartT、traits、Allocator> クラスのオブジェクトの読み取りをサポートする
wofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
wostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
wostringstream	basic_string<chart、traits、Allocator> クラスのオブジェクトの書き込みをサポートする
wstreambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
wstring	basic_string<wchar_t、char_traits<wchar_t>、allocator<wchar_t>> 用の型定義
wstringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける

STLport

libCstd の代替ライブラリを使用する場合は、標準ライブラリの STLport 実装を使用します。libCstd をオフにして、STLport ライブラリで代用するには、次のコンパイラオプションを使用します。

- -library=stlport4

詳細については、294 ページの「-library=l[,l...]」を参照してください。

このリリースでは、libstlport.a という静的アーカイブおよび libstlport.so という動的ライブラリの両方が含まれています。

STLport 実装を使用するかどうかは、以下を考慮して判断してください。

- STLport は、オープンソースの製品で、リリース間での互換性は保証されません。つまり、将来のバージョンの STLport でコンパイルすると、STLport 4.5.3 でコンパイルしたアプリケーションで問題が発生する可能性があります。また、STLport 4.5.3 でコンパイルしたバイナリは、将来のバージョンの STLport でコンパイルしたバイナリとリンクできない可能性があります。

- stlport4、Cstd、および iostream のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作は予期しないものになる恐れがあります。

- コンパイラの将来のリリースには、STLport4 が含まれない可能性があります。STLport の新しいバージョンだけが含まれる可能性があります。コンパイラオプションの -library=stlport4 は、将来のリリースでは使用できず、STLport のそれ以降のバージョンを示すオプションに変更される可能性があります。

- Tools.h++ は、STLport ではサポートされていません。

- STLport は、デフォルトの libCstd とはバイナリ互換ではありません。STLport の標準ライブラリの実装を使用する場合は、-library=stlport4 オプションを指定してすべてのファイルのコンパイルおよびリンクを実行する必要があります。このことは、たとえば STLport 実装と C++ 区間演算ライブラリ libCsunimath を同時に使用できないことを意味します。その理由は、libCsunimath のコンパイルに使用されたのが、STLport ではなくデフォルトライブラリヘッダーであるためです。

- STLport 実装を使用する場合は、コードから暗黙に参照されるヘッダーファイルをインクルードしてください。標準のヘッダーは、実装の一部として相互にインクルードすることができます (必須ではありません)。
- `-compat=4` によってコンパイルする場合には、STLport 実装を使用できません。

次の例は、ライブラリの実装について移植性のない想定が行われているため、STLport を使用してコンパイルできません。特に、`<vector>` または `<iostream>` が `<iterator>` を自動的にインクルードすることを想定していますが、これは正しい想定ではありません。

```
#include <vector>
#include <iostream>

using namespace std;

int main ()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

問題を解決するには、ソースで `<iterator>` をインクルードします。

第14章

従来の iostream ライブラリの使用

C++ も C と同様に組み込み型の入出力文はありません。その代わりに、出力機能はライブラリで提供されています。C++ コンパイラではiostream クラスに対して、従来型の実装と ISO 標準の実装を両方とも提供しています。

- 互換モード (-compat[=4]) では、従来型の iostream クラスは libC に含まれています。
- 標準モード (デフォルトのモード) では、従来型の iostream クラスは libiostream に含まれています。従来型の iostream クラスを使用したソースコードを標準モードでコンパイルするときは、libiostream を使用します。従来型の iostream の機能を標準モードで使用するには、iostream.h ヘッダーファイルをインクルードし、-library=iostreamオプションを使用してコンパイルします。
- 標準の iostream クラスは標準モードだけで使用でき、C++ 標準ライブラリ libCstd に含まれています。

この章では、従来型の iostream ライブラリの概要と使用例を説明します。この章では、iostream ライブラリを完全に説明しているわけではありません。詳細は、iostream ライブラリのマニュアルページを参照してください。従来型の iostream のマニュアルページを表示するには、次のように入力します (name にはマニュアルページのトピック名を入力)。

```
example% man -s 3CC4 name
```

定義済みの iostream

定義済みの iostream には、次のものがあります。

- cin、標準入力と結合しています。
- cout、標準出力と結合しています。
- cerr、標準エラーと結合しています。
- clog、標準エラーと結合しています。

定義済み iostreams は、cerr を除いて完全にバッファリングされています。198 ページの「iostream を使用した出力」 と 202 ページの「iostream を使用した入力」 を参照してください。

iostream 操作の基本構造

iostream ライブラリを使用すると、プログラムで必要な数の入出力ストリームを使用することができます。それぞれのストリームは、次のどれかを入力先または出力先とします。

- 標準入力
- 標準出力
- 標準エラー
- ファイル
- 文字型配列

ストリームは、入力のみまたは出力のみと制限して使用することも、入出力両方に使用することもできます。iostream ライブラリでは、次の 2 つの処理階層を使用してこのようなストリームを実現しています。

- 下層では、単なる文字ストリームであるシーケンスを実現します。シーケンスは、streambuf クラスか、その派生クラスで実現されています。
- 上層では、シーケンスに対してフォーマット操作を行います。フォーマット操作は istream と ostream の 2 つのクラスで実現されます。これらのクラスはメンバーに streambuf クラスから派生したオブジェクトを持っています。このほかに、入出力両方が実行されるストリームに対しては iostream クラスがあります。

標準入力、標準出力、標準エラーは、istream または ostream から派生した特殊なクラスオブジェクトで処理されます。

ifstream、ofstream、fstream の 3 つのクラスはそれぞれ istream、ostream、iostream から派生しており、ファイルへの入出力を処理します。

istrstream、ostrstream、strstream の 3 つのクラスはそれぞれ istream、ostream、および iostream から派生しており、文字型配列への入出力を処理します。

入力ストリームまたは出力ストリームをオープンする場合は、どれかの型のオブジェクトを生成し、そのストリームのメンバー streambuf をデバイスまたはファイルに関連付けます。通常、関連付けはストリームコンストラクタで行うので、ユーザーが直接 streambuf を操作することはありません。標準入力、標準出力、エラー出力に対しては、iostream ライブラリであらかじめストリームオブジェクトを定義してあるので、これらのストリームについてはユーザーが独自にオブジェクトを生成する必要はありません。

ストリームへのデータの挿入 (出力)、ストリームからのデータの抽出 (入力)、挿入または抽出したデータのフォーマット制御には、演算子または iostream のメンバー関数を使用します。

新たなデータ型 (ユーザー定義のクラス) を挿入したり抽出したりするときは一般に、挿入演算子と抽出演算子の多重定義をユーザーが行います。

従来型の iostream ライブラリの使用

従来型の iostream ライブラリルーチンを使用するには、ライブラリの使用部分に対応するヘッダーファイルをインクルードしなければなりません。次の表で各ヘッダーファイルについて説明します。

表 14-1 iostream ルーチンのヘッダーファイル

ヘッダーファイル	内容
iostream.h	iostream ライブラリの基本機能の宣言。
fstream.h	ファイルに固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。
strstream.h	文字型配列に固有の iostream と streambuf の宣言。この中で iostream.h をインクルードします。

表 14-1 iostream ルーチンのヘッダーファイル (続き)

ヘッダーファイル	内容
iomanip.h	マニピュレータ値の宣言。マニピュレータ値とは iostream に挿入または iostream から抽出する値で、特別の効果を引き起こします。この中で iostream.h をインクルードします。
stdiostream.h	(旧形式) 標準入出力の FILE 使用のための iostream と streambuf の宣言。この中で iostream.h をインクルードします。
stream.h	この中でiostream.h、fstream.h、iomanip.h、stdiostream.hをインクルードします。(旧形式) C++ バージョン 1.2 の旧形式ストリームと互換性を保つための宣言。

これらのヘッダーファイルすべてをプログラムにインクルードする必要はありません。自分のプログラムで必要な宣言の入ったものだけをインクルードします。互換モード (-compat[=4]) では、従来型の iostream ライブラリは libC の一部であり、CC ドライバによって自動的にリンクされます。標準モード (デフォルトのモード) では、従来型の iostream ライブラリは libiostream に含まれています。

iostream を使用した出力

iostream を使用した出力は、通常、左シフト演算子 (<<) を多重定義したもの (iostream の文脈では挿入演算子といいます) を使用します。ある値を標準出力に出力するには、その値を定義済みの出力ストリーム cout に挿入します。たとえば someValue を出力するには、次の文を標準出力に挿入します。

```
cout << someValue;
```

挿入演算子は、すべての組み込み型について多重定義されており、someValue の値は適当な出力形式に変換されます。たとえば someValue が float 型の場合、<< 演算子はその値を数字と小数点の組み合わせに変換します。float 型の値を出力ストリームに挿入するときは、<< を float 型挿入子といいます。一般に X 型の値を出力ストリームに挿入するときは、<< を X 型挿入子といいます。出力形式とその制御方法については、ios(3CC4) のマニュアルページを参照してください。

iostream ライブラリでは、ユーザー定義型については検知しません。したがってユーザー定義型を出力したい場合は、ユーザーが自分で挿入子を正しく定義する、すなわち << 演算子を多重定義する必要があります。

<< 演算子は反復使用することができます。2 つの値を cout に挿入するには、次の例のような文を使用することができます。

```
cout << someValue << anotherValue;
```

上の例では、2 つの値の間に空白が入りません。空白を入れたい場合は、次のようにします。

```
cout << someValue << " " << anotherValue;
```

<< 演算子は、組み込みの左シフト演算子と同じ優先順位を持ちます。他の演算子と同様に、括弧を使用して実行順序を指定することができます。実行順序をはっきりさせるためにも、括弧を使用するとよい場合がよくあります。次の 4 つの文のうち、最初の 2 つは同じ結果になりますが、後の 2 つは異なります。

```
cout << a+b;       // + は << より優先順位が高い
cout << (a+b);
cout << (a&y);     // << は & より優先順位が高い
cout << a&y;       // (cout <<a) & y となっておそらくエラーになる
```

ユーザー定義の挿入演算子

次のコーディング例では string クラスを定義しています。

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
    char* data;
    size_t size;

public:
    //(さまざまな関数定義)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

この例では、挿入演算子と抽出演算子をフレンド定義しておく必要があります。string クラスのデータ部が非公開だからです。

```
ostream& operator<< (ostream& ostr, const string& output)
{   return ostr << output.data; }
```

上は、string クラスに対して多重定義された演算子関数 operator<< の定義です。

```
cout << string1 << string2;
```

operator<< は、最初の引数として ostream& (ostream への参照) を受け取り、同じ ostream を返します。このため、次のように 1 つの文で挿入演算子を続けて使用することができます。

出力エラーの処理

operator<< を多重定義するときは、iostream ライブラリからエラーが通知されることになるため、特にエラー検査を行う必要はありません。

エラーが起こると、エラーの起こった iostream はエラー状態になります。その iostream の状態の各ビットが、エラーの大きな分類に従ってセットされます。iostream で定義された挿入子がストリームにデータを挿入しようとしても、そのストリームがエラー状態の場合はデータが挿入されず、そのストリームの状態も変わりません。

一般的なエラー処理方法は、メインのどこかで定期的に出力ストリームの状態を検査する方法です。そこで、エラーが起こっていることがわかれば、何らかの処理を行います。この章では、文字列を出力してプログラムを中止させる関数 error をユーザーが定義しているものとして説明します。関数 error はユーザー定義の関数で、定義済みの関数ではありません。関数 error の内容については、205 ページの「入力エラーの処理」を参照してください。iostream の状態を調べるには、演算子 !を使用します。次の例に示すように、iostream がエラー状態の場合はゼロ以外の値を返します。例を次に示します。

```
if (!cout) error( "output error");
```

エラーを調べるにはもう 1 つの方法があります。ios クラスでは、operator void *() が定義されており、エラーが起こった場合は NULL ポインタを返します。したがって、次の文でエラーを検査することができます。

```
if (cout << x) return ; // 正常終了のときのみ返す
```

また、次のように ios クラスのメンバー関数 good を使用することもできます。

```
if ( cout.good() ) return ; // 正常終了のときのみ返す
```

エラービットは次のような列挙型で宣言されています。

```
enum io_state { goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80} ;
```

エラー関数の詳細については、iostream のマニュアルページを参照してください。

出力のフラッシュ

多くの入出力ライブラリと同様、iostream も出力データを蓄積し、より大きなブロックにまとめて効率よく出力します。出力バッファをフラッシュしたければ、次のように特殊な値 flush を挿入するだけで、フラッシュすることができます。例を次に示します。

```
cout << "This needs to get out immediately." << flush ;
```

flush は、マニピュレータと呼ばれるタイプのオブジェクトの 1 つです。マニピュレータを iostream に挿入すると、その値が出力されるのではなく、何らかの効果が引き起こされます。マニピュレータは実際には関数で、ostream& または istream& を引数として受け取り、そのストリームに対する何らかの動作を実行した後にその引数を返します (212 ページの「マニピュレータ」を参照してください)。

バイナリ出力

ある値をバイナリ形式のままで出力するには、次の例のようにメンバー関数 write を使用します。次の例では、x の値がバイナリ形式のまま出力されます。

```
cout.write((char*)&x, sizeof(x));
```

この例では、&x を char* に変換しており、型変換の規則に反します。通常このようにしても問題はありませんが、x の型が、ポインタ、仮想メンバー関数、またはコンストラクタの重要な動作を要求するものを持つクラスの場合、上の例で出力した値を正しく読み込むことができません。

iostream を使用した入力

iostream を使用した入力は、iostream を使用した出力と同じです。入力には、抽出演算子>> を使用します。挿入演算子と同様に繰り返し指定することができます。例を次に示します。

```
cin >> a >> b ;
```

この例では、標準入力から 2 つの値が取り出されます。他の多重定義演算子と同様に、使用される抽出子の機能は a と b の型によって決まります (a と b の型が異なれば、別の抽出子が使用されます)。入力データのフォーマットとその制御方法についての詳細は、ios(3CC4) のマニュアルページを参照してください。通常は、先頭の空白文字 (スペース、改行、タブ、フォームフィードなど) は無視されます。

ユーザー定義の抽出演算子

ユーザーが新たに定義した型のデータを入力するには、出力のために挿入演算子を多重定義したのと同様に、その型に対する抽出演算子を多重定義します。

クラス string の抽出演算子は次のコーディング例のように定義します。

コード例 14-1 string の抽出演算子

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
```

コード例 14-1 string の抽出演算子 (続き)

```
    char holder[maxline];
    istr.get(holder, maxline, '\n');
    input = holder;
    return istr;
}
```

get 関数は、入力ストリーム istr から文字列を読み取ります。読み取られた文字列は、maxline-1 バイトの文字が読み込まれる、新しい行に達する、EOF に達する、のうちのいずれかが発生するまで、holder に格納されます。データ holder は NULL で終わります。最後に、holder 内の文字列がターゲットの文字列にコピーされます。

規則に従って、抽出子は第 1 引数 (上の例では istream& istr) から取り出した文字列を変換し、常に参照引数である第 2 引数に格納し、第 1 引数を返します。抽出子とは、入力値を第 2 引数に格納するためのものなので、第 2 引数は必ず参照引数でなければなりません。

char*の抽出子

この定義済み抽出子は問題が起こる可能性があるため、ここで説明しておきます。この抽出子は次のように使用します。

```
char x[50];
cin >> x;
```

上の例で、抽出子は先頭の空白を読み飛ばし、次の空白文字までの文字列を抽出して x にコピーします。次に、文字列の最後を示す NULL 文字 (0) を入れて文字列を完成します。ここで、入力文字列が指定した配列からあふれる可能性があることに注意してください。

さらに、ポインタが、割り当てられた記憶領域を指していることを確認する必要があります。次に示すのは、よく発生するエラーの例です。

```
char * p;// 初期化されていない
cin >> p;
```

入力データが格納される場所が特定されていません。これによって、プログラムが異常終了することがあります。

1 文字の読み込み

char 型の抽出子を使用することに加えて、次に示すいずれかの形式でメンバー関数 get を使用することによって、1 文字を読み取ることができます。例を次に示します。

```
char c;
cin.get(c); // 入力に失敗した場合は、cは変更なし

int b;
b = cin.get(); // 入力に失敗した場合は、b を EOF に設定
```

注 – 他の抽出子とは異なり、char 型の抽出子は行頭の空白を読み飛ばしません。

空白だけを読み飛ばして、タブや改行などその他の文字を取り出すようにするには、次のようにします。

```
int a;
do {
    a = cin.get();
   }
while( a == ' ' );
```

バイナリ入力

メンバー関数 write で出力したようなバイナリの値を読み込むには、メンバー関数 read を使用します。次の例では、メンバー関数 read を使用して x のバイナリ形式の値をそのまま入力します。次の例は、先に示した関数 write を使用した例と反対のことを行います。

```
cin.read((char*)&x, sizeof(x));
```

入力データの先読み

メンバー関数 peek を使用するとストリームから次の文字を抽出することなく、その文字を知ることができます。例を次に示します。

```
if (cin.peek() != c) return 0;
```

空白の抽出

デフォルトでは、iostream の抽出子は先頭の空白を読み飛ばします。*skip* フラグをオフにすれば、先頭の空白を読み飛ばさないようにすることができます。次の例では、cin の先頭の空白の読み飛ばしをいったんオフにし、後にオンに戻しています。

```
cin.unsetf(ios::skipws);     //先頭の空白の読み飛ばしをオフに設定
. . .
cin.setf(ios::skipws);       // 先頭の空白の読み飛ばしをオンに再設定
```

iostream のマニピュレータ ws を使用すると、空白の読み飛ばしが現在オンかオフかに関係なく、iostream から先頭の空白を取り除くことができます。次の例では、iostream istr から先頭の空白が取り除かれます。

```
istr >> ws;
```

入力エラーの処理

通常は、第 1 引数が非ゼロのエラー状態にある場合、抽出子は入力ストリームからのデータの抽出とエラービットのクリアを行わないでください。データの抽出に失敗した場合、抽出子は最低 1 つのエラービットを設定します。

出力エラーの場合と同様、エラー状態を定期的に検査し、非ゼロの状態の場合は処理の中止など何らかの動作を起こす必要があります。演算子！は、iostream のエラー状態を検査します。たとえば次のコーディング例では、英字を入力すると入力エラーが発生します。

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n" ;
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

クラス ios には、エラー処理に使用できるメンバー関数があります。詳細はマニュアルページを参照してください。

iostream と stdio の併用

C++ でも stdio を使用することができますが、プログラムで iostream と stdio とを標準ストリームとして併用すると、問題が起こる場合があります。たとえば stdout と cout の両方に書き込んだ場合、個別にバッファリングされるため出力結果が設計したとおりにならないことがあります。stdin と cin の両方から入力した場合、問題はさらに深刻です。個別にバッファリングされるため、入力データが使用できなくなってしまいます。

標準入力、標準出力、標準エラーに関するこのような問題を解決するためには、入出力に先立って次の命令を実行します。次の命令で、すべての定義済み iostream が、それぞれ対応する定義済みの標準入出力の FILE に結合されます。

```
ios::sync_with_stdio();
```

このような結合を行うと、定義済みストリームが結合されたものの一部となってバッファリングされなくなってかなり効率が悪くなるため、デフォルトでは結合されていません。同じプログラムでも、stdio と iostream を別のファイルに対して使用することはできます。すなわち、stdio ルーチンを使用して stdout に書き込み、iostream に結合した別のファイルに書き込むことは可能です。また stdio FILE を入力用にオープンしても、stdin から入力しない限りは cin からも読み込むことができます。

iostreamの作成

定義済みの iostream 以外のストリームへの入出力を行いたい場合は、ユーザーが自分で iostream を生成する必要があります。これは一般には、iostream ライブラリで定義されている型のオブジェクトを生成することになります。ここでは、使用できるさまざまな型について説明します。

クラス fstream を使用したファイル操作

ファイル操作は標準入出力の操作に似ています。ifstream、ofstream、fstream の 3 つのクラスはそれぞれ、istream、ostream、iostream の各クラスから派生しています。この 3 つのクラスは派生クラスなので、挿入演算と抽出演算、および、その他のメンバー関数を継承しており、ファイル使用のためのメンバーとコンストラクタも持っています。

fstream のいずれかを使用するときは、fstream.h をインクルードしなければなりません。入力だけ行うときは ifstream、出力だけ行うときは ofstream、入出力を行うときは fstream を使用します。コンストラクタへの引数としてはファイル名を渡します。

thisFile というファイルから thatFile というファイルへのファイルコピーを行うときは、次のコーディング例のようになります。

```
ifstream fromFile("thisFile");
if  (!fromFile)
    error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if  ( !toFile )
    error("unable to open 'thatFile' for output");
char c ;
while (toFile && fromFile.get(c)) toFile.put(c);
```

このコードでは次のことを実行します。

- fromFile という ifstream オブジェクトをデフォルトモード ios::in で生成し、それを thisFile に結合します。thisFile をオープンします。
- 生成した ifstream オブジェクトのエラー状態を調べ、エラーであれば関数 error を呼び出します。関数 error は、プログラムのどこか別のところで定義されている必要があります。
- toFile という ofstream オブジェクトをデフォルトモード ios::out で生成し、それを thatFile に結合します。
- fromFile と同様に、toFile のエラー状態を検査します。
- データの受け渡しに使用する char 型変数を生成します。
- fromFile の内容を一度に 1 文字ずつ toFile にコピーします。

注 – ファイルの内容を一度に 1 文字ずつコピーすることは実際にはあまり行われません。このコードは fstream の使用例として示したにすぎません。実際には、入力ストリームに関係付けられた streambuf を出力ストリームに挿入するのが一般的です。217 ページの「Streambufs」 と sbufpub(3CC4) のマニュアルページを参照してください。

オープンモード

オープンモードは、列挙型 open_mode の各ビットの OR をとって設定します。open_mode は、ios クラスの公開部で次のように定義されています。

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

注 – UNIX では binary フラグは必要ありませんが、binary フラグを必要とするシステムとの互換性を保つために提供されています。移植可能なコードにするためには、バイナリファイルをオープンするときに binary フラグを使用する必要があります。

入出力両用のファイルをオープンすることができます。たとえば次のコードでは、someName という入出力ファイルをオープンして、fstream 変数 inoutFile に結合します。

```
fstream inoutFile("someName", ios::in|ios::out);
```

ファイルを指定しない fstream の宣言

ファイルを指定せずに fstream の宣言だけを行い、後にファイルをオープンすることもできます。次の例では出力用のofstream toFile を作成します。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

ファイルのオープンとクローズ

fstream をいったんクローズし、また別のファイルでオープンすることができます。たとえば、コマンド行で与えられるファイルリストを処理するには次のようにします。

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

ファイル記述子を使用したファイルのオープン

標準出力は整数 1 などのようにファイル記述子がわかっている場合は、次のようにファイルをオープンすることができます。

```
ofstream outfile;
outfile.attach(1);
```

fstream のコンストラクタにファイル名を指定してオープンしたり、open 関数を使用してオープンしたファイルは、fstream が破壊された時点 (delete するか、スコープ外に出る時点) で自動的にクローズされます。attach で fstream に結合したファイルは、自動的にはクローズされません。

ファイル内の位置の再設定

ファイル内の読み込み位置と書き込み位置を変更することができます。そのためには次のようなツールがあります。

- streampos は、iostream 内の位置を記憶しておくためのデータ型です。
- tellg (tellp) は istream (ostream) のメンバー関数で、現在のファイル内の位置を返します。istream と ostream は fstream の親クラスですので、tellg と tellp も fstream クラスのメンバー関数として呼び出すことができます。
- seekg (seekp) は istream (ostream) のメンバー関数で、指定したファイル内の位置を探し出します。

■ 列挙型 seek_dir は、seek での相対位置を指定します。

```
enum seek_dir { beg=0, cur=1, end=2 };
```

fstream aFile の位置再設定の例を次に示します。

```
streampos original = aFile.tellp(); //現在の位置の保存
aFile.seekp(0, ios::end); //ファイルの最後に位置を再設定
aFile << x;               //データをファイルに書き込む
aFile.seekp(original);    //元の位置に戻る
```

seekg（seekp）は、1 つまたは 2 つの引数を受け取ります。引数を 2 つ受け取るときは、第 1 引数は、第 2 引数で指定した seek_dir 値が示す位置からの相対位置となります。例を次に示します。

```
aFile.seekp(-10, ios::end);
```

この例では、ファイルの最後から 10 バイトの位置に設定されます。

```
aFile.seekp(10, ios::cur);
```

一方、次の例では現在位置から 10 バイト進められます。

注 – テキストストリーム上での任意位置へのシーク動作はマシン依存になります。ただし、以前に保存した streampos の値にいつでも戻ることができます。

---

iostream の代入

iostream では、あるストリームを別のストリームに代入することはできません。

ストリームオブジェクトをコピーすると、出力ファイル内の現在の書き込み位置ポインタなどの位置情報が二重に存在するようになり、それを個別に変更できるという状態が起こります。これは、ストリーム操作を混乱させる可能性があります。

フォーマットの制御

フォーマットの制御については、ios (3CC4) のマニュアルページで詳しく説明しています。

マニピュレータ

マニピュレータとは、iostream に挿入したり、iostream から抽出したりする値で特別な効果を持つもののことです。

引数付きマニピュレータとは、1 つ以上の追加の引数を持つマニピュレータのことです。

マニピュレータは通常の識別子であるため、マニピュレータの定義を多く行うと可能な名前を使いきってしまうので、iostream では考えられるすべての機能に対して定義されているわけではありません。マニピュレータの多くは、この章の別の箇所でメンバー関数とともに説明しています。

定義済みマニピュレータは 13 個あり、それぞれについては表 14-2 で説明します。この表で使用している文字の意味は次のとおりです。

- i は long 型です。
- n は int 型です。
- c は char 型です。
- istr は 入力ストリームです。
- ostr は 出力ストリームです。

表 14-2 iostream の定義済みマニピュレータ

	定義済みマニピュレータ	内容
1	`ostr << dec, istr >> dec`	基数が 10 の整数変換を指定します。
2	`ostr << endl`	復帰改行文字('\n') を挿入して、ostream::flush() を呼び出します。
3	`ostr << ends`	NULL (0) 文字を挿入。strstream 使用時に利用します。
4	`ostr << flush`	ostream::flush() を呼び出します。
5	`ostr << hex,  istr >> hex`	基数が 16 の整数変換を指定します。
6	`ostr << oct,  istr >> oct`	基数が 8 の整数変換を指定します。
7	`istr >> ws`	最初に空白以外の文字が見つかるまで (この文字以降は istr に残る)、空白を取り除きます (空白を読み飛ばす)。
8	`ostr << setbase(n),` `istr >> setbase(n)`	基数が n (0, 8, 10, 16 のみ) の整数変換を指定します。
9	`ostr << setw(n),  istr >> setw(n)`	ios::width(n) を呼び出します。フィールド幅を n に設定します。
10	`ostr << resetiosflags(i),` `istr >> resetiosflags(i)`	i のビットセットに従って、フラグのビットベクトルをクリアします。
11	`ostr << setiosflags(i),` `istr >> setiosflags(i)`	iのビットセットに従って、フラグのビットベクトルを設定します。
12	`ostr << setfill(c),` `istr >> setfill(c)`	埋め込み文字 (フィールドのパディング用文字) を c とします。
13	`ostr << setprecision(n),` `istr >> setprecision(n)`	浮動小数点型データの精度を n 桁にします。

定義済みマニピュレータを使用するには、プログラムにヘッダーファイルiomanip.hをインクルードする必要があります。

ユーザーが独自のマニピュレータを定義することもできます。マニピュレータには次の 2 つの基本タイプがあります。

- 引数なしのマニピュレータistream&、ostream&、ios& のどれかを引数として受け取り、ストリームの操作が終わるとその引数を返します。
- 引数付きのマニピュレータistream&、ostream&、ios& のどれかと、その他もう 1 つの引数 (追加の引数) を受け取り、ストリームの操作が終わるとストリーム引数を返します。以下に、それぞれのタイプのマニピュレータの例を示します。

引数なしのマニピュレータの使用法

引数なしのマニピュレータは、次の 3 つを実行する関数です。

- ストリームの参照引数を受け取ります。
- そのストリームに何らかの処理を行います。
- その引数を返します。

iostream では、このような関数 (へのポインタ) を使用するシフト演算子がすでに定義されていますので、関数を入出力演算子シーケンスの中に入れることができます。シフト演算子は、値の入出力を行う代わりに、その関数を呼び出します。tab を ostream に挿入する tab マニピュレータの例を示します。

```
ostream& tab(ostream& os) {
         return os << '\t' ;
         }
...
cout << x << tab << y ;
```

次のコードは、上の例と同じ処理をより洗練された方法で行います。

```
const char tab = '\t';
...
cout << x << tab << y;
```

次に示すのは別の例で、定数を使用してこれと同じことを簡単に実行することはできません。入力ストリームに対して、空白の読み飛ばしのオン、オフを設定したいと仮定します。ios::setf と ios::unsetf を別々に呼び出して、skipws フラグをオンまたはオフに設定することもできますが、次の例のように 2 つのマニピュレータを定義して設定することもできます。

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
        is.setf(ios::skipws, ios::skipws);
        return is;
}
istream& skipoff(istream& is) {
        is.unsetf(ios::skipws);
        return is;
}
...
int main ()
{
       int x,y;
       cin >> skipon >> x >> skipoff >> y;
       return 1;
}
```

引数付きのマニピュレータの使用法

iomanip.h に入っているマニピュレータの 1 つに setfill があります。setfill は、フィールド幅に詰め合わせる文字を設定するマニピュレータで、次の例に示すように定義されています。

```
//ファイル setfill.cc
#include<iostream.h>
#include<iomanip.h>

//非公開のマニピュレータ
static ios& sfill(ios& i, int f) {
          i.fill(f);
          return i;
}
//公開の適用子
smanip_int setfill(int f) {
        return smanip_int(sfill, f);
}
```

引数付きマニピュレータは、2 つの部分から構成されます。

- 1 つはマニピュレータ部分で、これは引数を 1 つ追加します。この前の例では、int 型の第 2 引数があります。このような関数に対するシフト演算子は定義されていませんので、このマニピュレータ関数を入出力演算子シーケンスに入れることはできません。そこで、マニピュレータの代わりに補助関数 (適用子) を使用する必要があります。
- もう 1 つは適用子で、これはマニピュレータを呼び出します。適用子は大域関数で、そのプロトタイプをヘッダーファイルに入れておきます。マニピュレータは通常、適用子の入っているソースコードファイル内に静的関数として作成します。マニピュレータは適用子からのみ呼び出されるので、静的関数にして、大域アドレス空間にマニピュレータ関数名を入れないようにします。

ヘッダーファイル iomanip.h には、さまざまなクラスが定義されています。各クラスには、マニピュレータ関数のアドレスと 1 つの引数の値が入っています。iomanip クラスについては、manip (3CC4) のマニュアルページで説明しています。この前の例では、smanip_int クラスを使用しており、ios で使用できます。ios で使用できるということは、istream と ostream でも使用できるということです。この例ではまた、int 型の第 2 引数を使用しています。

適用子は、クラスオブジェクトを作成してそれを返します。この前の例では、smanip_int というクラスオブジェクトが作成され、そこにマニピュレータと、適用子の int 型引数が入っています。ヘッダーファイル iomanip.h では、このクラスに対するシフト演算子が定義されています。入出力演算子シーケンスの中に適用子関数 setfill があると、その適用子関数が呼び出され、クラスが返されます。シフト演算子はそのクラスに対して働き、クラス内に入っている引数値を使用してマニピュレータ関数が呼び出されます。

次の例では、マニピュレータ print_hex は以下のことを行います。

- 出力ストリームを 16 進モードする。
- long 型の値をストリームに挿入する。
- ストリームの変換モードを元に戻す。

この例は出力専用のため、omanip_long クラスが使用されています。また、int 型でなく long 型でデータを操作します。

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

Strstreams：配列用の iostreams

strstream(3CC4) のマニュアルページを参照してください。

Stdiobufs：標準入出力ファイル 用の iostream

stdiobuf(3CC4) のマニュアルページを参照してください。

Streambufs

入力や出力のシステムは、フォーマットを行う iostream と、フォーマットなしの文字ストリームの入力または出力を行う streambuf からなります。

通常は iostream を通して streambuf を使用するので、streambuf の詳細を知る必要はありません。ただし、効率をよくするため、または iostream に組み込まれているエラー処理やフォーマットのためなどに必要な場合は、直接 streambuf を使用することができます。

streambuf の機能

streambuf は文字シーケンス (文字ストリーム) と、シーケンス内を指す 1 つまたは 2 つのポインタとで構成されています。各ポインタは文字と文字の間を指しています。実際には文字と文字の間を指しているわけではありませんが、このように考えておくと理解しやすくなります。streambuf ポインタには次の種類があります。

- *put* ポインタ次に streambuf から渡す文字の直前を指します。
- *get* ポインタ次に streambuf から取り出す文字の直前を指します。

streambuf は、このどちらかのポインタ、または両方のポインタを持ちます。

ポインタの位置

ポインタ位置の操作とシーケンスの内容の操作にはさまざまな方法があります。文字列の操作時に両方のポインタが移動するかどうかは、使用される streambuf の種類によって違います。一般に、キュー形式の streambuf の場合は、get ポインタと put ポインタは別々に移動し、ファイル形式の streambuf の場合は、get ポインタと put ポインタは同時に移動します。キュー形式ストリームの例としては strstream があり、ファイル形式ストリームの例としては fstream があります。

streambuf の使用

ユーザーは streambuf オブジェクト自体を作成することはなく、streambuf クラスから派生したクラスのオブジェクトを作成します。その例として、filebuf と strstreambuf とがあります。この 2 つについてはそれぞれ filebuf (3CC4) および ssbuf (3) のマニュアルページを参照してください。より高度な使い方として、独自のクラスを streambuf から派生させて特殊デバイスのインタフェースを提供したり、基本的なバッファリング以外のバッファリングを行なったりすることができます。sbufpub (3CC4) と sbufprot (3CC4) のマニュアルページでは、それらの方法について説明しています。

ユーザー用の特殊な streambuf を作成するとき以外にも、上に示したマニュアルページで説明しているように、iostream と結合した streambuf にアクセスして公開メンバー関数を使用したい場合があります。また、各 iostream には、streambuf へのポインタを引数とする定義済みの挿入子と抽出子があります。streambuf を挿入したり抽出したりすると、ストリーム全体がコピーされます。

次の例では、先に説明したファイルコピーとは違う方法でファイルをコピーしています。簡単にするため、エラー検査は省略しています。

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

入力ファイルと出力ファイルは、以前の例と同じ方法でオープンします。各 iostream クラスにはメンバー関数 rdbuf があり、それに結合した streambuf オブジェクトへのポインタを返します。fstream の場合、streambuf オブジェクトは filebuf 型です。fromFile に結合したファイル全体が toFile に結合したファイルにコピー (挿入) されます。最後の行は次のように書くこともできます。

```
fromFile >> toFile.rdbuf();
```

上の書き方では、ソースファイルが抽出されて目的のところに入ります。どちらの書き方をしても、結果はまったく同じになります。

iostream に関するマニュアルページ

C++ では、iostream ライブラリの詳細を説明する多くのマニュアルページがあります。次に、各マニュアルページの概要を示します。

従来型の iostream ライブラリのマニュアルページを表示するには、次のように入力します (*name* には、マニュアルページのトピック名を入力)。

```
example% man -s 3CC4 name
```

表 14-3 iostream に関するマニュアルページの概要

マニュアルページ	概要
filebuf	streambuf から派生し、ファイル処理のために特殊化された filebuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承した機能の詳細については、sbufpub(3CC4) と sbufprot(3CC4) のマニュアルページを参照してください。filebuf クラスは、fstream クラスを通して使用します。
fstream	istream、ostream、iostreamをファイル処理用に特殊化した ifstream、ofstream、fstream の各クラスの特殊化されたメンバー関数を詳細に説明します。
ios	iostreamの基底クラスである ios クラスの各部を詳細に説明します。すべてのストリームに共通の状態データについても説明します。
ios.intro	iostream を紹介し、概要を説明します。
istream	次の各項目を詳細に説明します。 • streambuf から取り出した文字の解釈をサポートする、クラス istream のメンバー関数 • 入力の書式設定 • ostream の一部として記述されている位置決め関数 • 一部の関連関数 • 関連マニピュレータ
manip	iostreamライブラリで定義されている入出力マニピュレータを説明します。
ostream	次の各項目を詳細に説明します。 • streambuf から取り出した文字の解釈をサポートする、クラス istream のメンバー関数 • 出力の書式設定 • ostream の一部として記述されている位置決め関数 • 一部の関連関数 • 関連マニピュレータ
sbufprot	streambuf(3CC4) クラスから派生したクラスをコーディングするプログラマに必要なインタフェースを説明します。公開関数のいくつかは、このマニュアルページでは説明しないため、sbufpub(3CC4) のマニュアルページも参照してください。

表 14-3 iostream に関するマニュアルページの概要 (続き)

マニュアルページ	概要
sbufpub	streambuf クラスの公開インタフェース、特に streambuf の公開メンバー関数について詳細に説明します。このマニュアルページには、streambuf 型のオブジェクトを直接操作したり、streambuf から派生したクラスが継承している関数を探し出したりするのに必要な情報が含まれています。streambuf からクラスを派生する場合は、sbufprot (3CC4) のマニュアルページも参照してください。
ssbuf	streambuf から派生し、文字型配列処理用に特殊化された strstreambuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承する機能の詳細については、sbufpub (3CC4) のマニュアルページを参照してください。
stdiobuf	streambuf から派生し、標準入出力の FILE 処理のために特殊化された stdiobuf クラスについて最小限の説明をします。streambuf クラスから継承する機能の詳細については、sbufpub (3CC4) のマニュアルページを参照してください。
strstream	strstream の特殊化されたメンバー関数を詳細に説明します。これらの関数は、iostream クラスから派生した一連のクラスで実装され、文字型配列処理用に特殊化されています。

iostream の用語

iostream ライブラリの説明では、一般のプログラミングに関する用語と同じでも意味が異なる語を多く使用します。次の表では、それらの用語が iostream ライブラリの説明で使用される場合の意味を定義します。

表 14-4 iostream の用語

iostream 用語	意味
バッファ	バッファには、2 つの意味があります。1 つは iostream パッケージに固有のバッファで、もう 1 つは入出力一般に適用されるバッファです。iostream ライブラリに固有のバッファは、streambuf クラスで定義されたオブジェクトです。 一般にいうバッファは、入出力データを効率よく転送するために使用するメモリーブロックを指します。バッファリングされた入出力の場合は、バッファがいっぱいになるか、バッファが強制的にフラッシュされるときまで、データの転送は行われません。 「バッファリングなしのバッファ」とは、上で定義した一般にいうバッファがない streambuf を指します。この章では streambuf を指すバッファという語を使用しないようにしていますが、マニュアルページや他の C++ のマニュアルでは、streambuf の意味でバッファという語を使用しています。
抽出	iostream から入力データを取り出す操作を抽出といいます。
Fstream	ファイル用に特殊化された入出力ストリームです。特に courier のようにクーリエフォントで印刷されている場合は、iostream クラスから派生した fstream クラスを指します。
挿入	iostream に出力データを送り込む操作を挿入といいます。
iostream	一般には、入力ストリームまたは出力ストリームです。
iostream ライブラリ	ファイルiostream.h、fstream.h、strstream.h、iomanip.h、ライブラリ stdiostream.h をインクルードすることにより使用できるライブラリです。iostream はオブジェクト指向のライブラリですので、ユーザーが必要に応じて拡張することができます。そのため、iostream ライブラリを使用して実行できるすべての機能があらかじめ定義されているわけではありません。

表 14-4 iostream の用語 (続き)

iostream 用語	意味
ストリーム	一般に、iostream、fstream、strstream、またはユーザー定義のストリームをいいます。
Streambuf	文字シーケンスの入ったバッファで、put ポインタまたは get ポインタ (またはその両方) を持ちます。courier のようにクーリエフォントで印刷されている場合は、streambuf という特定のクラスを意味します。その他のフォントで印刷されている場合は一般に streambuf クラスのオブジェクト、または streambuf の派生クラスを意味します。ストリームオブジェクトは必ず、streambuf から派生した型のオブジェクト (またはそのオブジェクトへのポインタ) を持っています。
Strstream	文字型配列処理用に特殊化した iostream です。courier のようにクーリエフォントで印刷されている場合は、strstream という特定のクラスを意味します。

第 15 章

複素数演算ライブラリの使用

下の例のように、複素数には「実部」と「虚部」があります。例を次に示します。

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

通常は、`0 + 3i` のように完全に虚部だけのものは通常 `3i` と書き、`5 + 0i` のように完全に実部だけのものは通常 `5` と書きます。データ型 `complex` を使用すると複素数を表現することができます。

注 – 複素数ライブラリ (`libcomplex`) は互換モードでのみ使用できます (`-compat[=4]`)。標準モード (デフォルトのモード) では、同様の機能を持つ複素数クラスが C++ 標準ライブラリ (`libCstd`) に含まれています。

複素数ライブラリ

複素数ライブラリは、新しいデータ型として複素数データ型を実装します。このライブラリには以下が含まれています。

- 演算子
- 数学関数 (組み込み数値型用に定義されている関数)
- 拡張機能 (複素数の入出力を可能にする iostream 用)
- エラー処理機能

複素数には、実部と虚部による表現方法の他に、絶対値と偏角による表現方法があります。複素数ライブラリには、実部と虚部によるデカルト表現と、絶対値と偏角による極座標表現とを互いに変換する関数も提供しています。

共役複素数は、虚部の符号が反対の複素数です。

複素数ライブラリの使用方法

複素数ライブラリを使用する場合は、プログラムにヘッダーファイル complex.h をインクルードし、-lcomplex オプションまたは -library=complex オプションを使用してリンクしてください。

complex 型

複素数ライブラリでは、クラス complex が 1 つだけ定義されています。クラス complex のオブジェクトは、1 つの複素数を持つことができます。複素数は次の 2 つの部分で構成されています。

- 実部
- 虚部

```
class complex {
    double re, im;
};
```

クラス complex のオブジェクトの値は、1 組の double 型の値です。最初の値が実部を表し、2 番目の値が虚部を表します。

complex クラスのコンストラクタ

complex には 2 つのコンストラクタがあります。それぞれの定義を次に示します。

```
complex::complex(){ re=0.0; im=0.0; }
complex::complex(double r, double i = 0.0)  { re=r; im=i; }
```

複素数の変数を引数なしで宣言すると、最初のコンストラクタが使用され、実部も虚部もゼロで初期化されます。次の例では、実部も虚部もゼロの複素数の変数が生成されます。

```
complex aComp;
```

1 つまたは 2 つのパラメータを指定できます。いずれの場合にも、2 番目のコンストラクタが使用されます。次の例のように、引数を 1 つだけ指定した場合は、その値は実部の値とみなされ虚部はゼロに設定されます。

```
complex aComp(4.533);
```

複素数の値は次のようになります。

```
4.533 + 0i
```

次の例のように、引数を 2 つ指定した場合は、最初の値が実部、2 番目の値が虚部となります。

```
complex aComp(8.999, 2.333);
```

複素数の値は次のようになります。

```
8.999 + 2.333i
```

また、複素数ライブラリが提供する polar 関数を使用して複素数を生成することもできます (228 ページの「数学関数」を参照してください)。polar 関数は、指定した 1 組の極座標値 (絶対値と偏角) を使用して複素数を作成します。

complex 型にはデストラクタはありません。

算術演算子

複素数ライブラリでは、すべての基本算術演算子が定義されています。特に、次の 5 つの演算子は通常の型の演算と同様に使用することができ、優先順序も同じです。

```
+   -   /   *   =
```

演算子 - は、通常の型の場合と同様に 2 項演算子としても単項演算子としても使用できます。

このほか、次の演算子の使用方法も通常の型で使用する演算子と同様です。

- 加算代入演算子 (+=)

- 減算代入演算子 (-=)
- 乗算代入演算子 (*=)
- 除算代入演算子 (/=)

ただし、この 4 つの演算子については、式の中で使用可能な値は生成されません。したがって、次のコードは機能しません。

```
complex a, b;
...
if ((a+=2)==0) {...}; // 誤り
b = a *= b; // 誤り
```

また、等しいか否かを判定する 2 つの演算子 (==、!=) は、通常の型で使用する演算子と同様に使用することができます。

算術式で実数と複素数が混在しているときは、C++ では複素数のための演算子関数が使用され、実数は複素数に変換されます。

数学関数

複素数ライブラリには、多くの数学関数が含まれています。複素数に特有のものもあれば、C の標準数学ライブラリの関数と同じで複素数を対象にしたものもあります。

これらの関数はすべて、あらゆる可能な引数に対して結果を返します。関数が数学的に正しい結果を返せないような場合は、`complex_error` を呼び出して、何らかの適切な値を返します。たとえば、オーバーフローが実際に起こるのを避けるために `complex_error` を呼び出してメッセージを出します。次の表で複素数ライブラリの関数を説明します。

注 – `sqrt` 関数と `atan2` 関数は、C99 の csqrt (Annex G) の仕様に従って実装されています。

表 15-1 複素数ライブラリの関数

複素数ライブラリ関数	内容
`double abs(const complex)`	複素数の絶対値を返します。
`double arg(const complex)`	複素数の偏角を返します。
`complex conj(const complex)`	引数に対する共役複素数を返します。
`double imag(const complex&)`	複素数の虚部を返します。
`double norm(const complex)`	引数の絶対値の 2 乗を返します。`abs` より高速ですが、オーバーフローが起きやすくなります。絶対値の比較に使用します。
`complex polar(double mag, double ang=0.0)`	複素数の絶対値と偏角を表す一組の極座標を引数として受け取り、それに対応する複素数を返します。
`double real(const complex&)`	複素数の実部を返します。

表 15-2 複素数の数学関数と三角関数

複素数ライブラリ関数	内容
`complex acos(const complex)`	引数が余弦となるような角度を返します。
`complex asin(const complex)`	引数が正弦となるような角度を返します。
`complex atan(const complex)`	引数が正接となるような角度を返します。
`complex cos(const complex)`	引数の余弦を返します。
`complex cosh(const complex)`	引数の双曲線余弦を返します。
`complex exp(const complex)`	`e**x` を計算します。ここで `e` は自然対数の底で、`x` は関数 `exp` に渡された引数です。
`complex log(const complex)`	引数の自然対数を返します。
`complex log10(const complex)`	引数の常用対数を返します。

表 15-2 複素数の数学関数と三角関数 (続き)

複素数ライブラリ関数	内容

`complex pow(double b, const complex exp)`
`complex pow(const complex b, int exp)`
`complex pow(const complex b, double exp)`
`complex pow(const complex b, const complex exp)`
	引数を 2 つ持ちます。`pow(`*b, exp*`)`.*b* を *exp*乗します。

`complex sin(const complex)`	引数の正弦を返します。
`complex sinh(const complex)`	引数の双曲線正弦を返します。
`complex sqrt(const complex)`	引数の平方根を返します。
`complex tan(const complex)`	引数の正接を返します。
`complex tanh(const complex)`	引数の双曲線正接を返します。

エラー処理

複素数ライブラリでは、エラー処理が次のように定義されています。

```
extern int errno;
class c_exception { ... };
int complex_error(c_exception&);
```

外部変数 `errno` は C ライブラリの大域的なエラー状態です。`errno` は、標準ヘッダー `errno.h` (perror(3) のマニュアルページを参照) にリストされている値を持ちます。`errno` には、多くの関数でゼロ以外の値が設定されます。

ある特定の演算でエラーが起こったかどうか調べるには、次のようにしてください。

1. **演算実行前に`errno` をゼロに設定する。**
2. **演算終了後に値を調べる。**

   関数 `complex_error` は `c_exception` 型の参照引数を持ち、次に示す複素数ライブラリ関数に呼び出されます。

- exp
- log
- log10
- sinh
- cosh

デフォルトの complex_error はゼロを返します。ゼロが返されたということは、デフォルトのエラー処理が実行されたということです。ユーザーは独自の complex_error 関数を作成して、別のエラー処理を行うことができます。エラー処理については、cplexrr(3CC4) のマニュアルページで説明しています。

デフォルトのエラー処理については、cplxtrig(3CC4) と cplxexp(3CC4) のマニュアルページを参照してください。次の表にも、その概要を掲載しています。

表 15-3 複素数ライブラリ関数のデフォルトエラー処理

複素数ライブラリ関数	デフォルトエラー処理
exp	オーバーフローが起こった場合は errno を ERANGE に設定し、最大複素数を返します。
log, log10	引数がゼロの場合は errno を EDOM に設定し、最大複素数を返します。
sinh, cosh	引数の虚部によりオーバーフローが起こる場合は複素数ゼロを返します。引数の実部によりオーバーフローが起こる場合は最大複素数を返します。どちらの場合も errno は ERANGE に設定されます。

入出力

複素数ライブラリでは、次の例に示す複素数のデフォルトの抽出子と挿入子が提供されています。

```
ostream& operator<<(ostream&, const complex&); //挿入子
istream& operator>>(istream&, complex&) //抽出子
```

抽出子と挿入子の基本的な説明については、196 ページの「iostream 操作の基本構造」 と 198 ページの「iostream を使用した出力」を参照してください。

入力の場合、複素数の抽出子 >> は、(括弧の中にあり、コンマで区切られた) 一組の値を入力ストリームから抽出し、複素数オブジェクトに読み込みます。最初の値が実部の値、2 番目の値が虚部の値となります。たとえば、次のような宣言と入力文がある場合、

```
complex x;
cin >> x;
```

(3.45, 5) と入力すると、複素数 x の値は3.45 + 5.0i となります。抽出子の場合はこの反対になります。complex x(3.45, 5)、cout<<x の場合は、(3.45, 5) と表示されます。

入力データは、通常括弧の中でコンマで区切られた一組の値で、スペースは入れても入れなくてもかまいません。値を 1 つだけ入力したとき (括弧とスペースは入力してもしなくても同じ) は、抽出子は虚部をゼロとします。シンボル i を入力してはいけません。

挿入子は、複素数の実部と虚部をコンマで区切り、全体を括弧で囲んで挿入します。シンボル i は含まれません。2 つの値は double 型として扱われます。

混合演算

complex 型は、組み込みの算術型と混在した式でも使用できるように定義されています。混合算術演算においては、算術型は自動的に complex 型に変換されます。算術演算子のすべてと数学関数のほとんどに対して、complex 型を使用できるバージョンが提供されています。例を次に示します。

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

b+i という式は混合算術演算です。整数 i は、コンストラクタ complex::complex(double,double=0) によって、complex 型に変換されます (このとき、まず整数から double 型に変換されます)。b+i の計算結果を double 型

の y で割っているので、y もまた complex 型に変換され、複素数除算演算が使用されます。商もまた complex 型ですので、複素数の正弦関数が呼び出され、その結果も complex 型になります。以下も同様です。

ただし、すべての算術演算と型変換が暗黙に行われるわけではありませんし、定義されていないものもあります。たとえば、複素数は数学的な意味での大小関係が決められないので、比較は等しいか否かの判定しかできません。

```
complex a, b;
a == b // OK
a != b // OK
a <  b // エラー：演算子 < は complex 型に使用できない
a >= b // エラー：演算子 >= は complex 型に使用できない
```

同様に、complex 型からそれ以外の型への変換もはっきりした定義ができないので、そのような自動変換は行われません。変換するときは、実部または虚部を取り出すのか、または絶対値を取り出すのかを指定する必要があります。

```
complex a;
double f(double);
f(abs(a)); // OK
f(a);      // エラー：f(complex) に一致するものがない
```

効率

クラス complex は効率も考慮して設計されています。

非常に簡単な関数が inline で宣言されており、関数呼び出しのオーバーヘッドをなくしています。

効率に差があるものは、関数が多重定義されています。たとえば、pow 関数には引数が complex 型のもののほかに、引数が double 型と int 型のものがあります。その方が double 型と int 型の計算がはるかに簡単になるからです。

complex.h をインクルードすると、C の標準数学ライブラリヘッダー math.h も自動的にインクルードされます。C++ の多重定義の規則により、次のようにもっとも効率の良い式の評価が行われます。

```
double x;
complex x = sqrt(x);
```

この例では、標準数学関数 sqrt(double) が呼び出され、その計算結果が complex 型に変換されます。最初に complex 型に変換され、sqrt(complex) が呼び出されるのではありません。これは、多重定義の解決規則から決まる方法で、もっとも効率の良い方法です。

複素数のマニュアルページ

複素数演算ライブラリの情報は、次のマニュアルページに記載されています。

表 15-4 complex 型のマニュアルページ

マニュアルページ	概要
cplx.intro(3CC4)	複素数ライブラリ全体の紹介
cartpol(3CC4)	直角座標と極座標の関数
cplxerr(3CC4)	エラー処理関数
cplxexp(3CC4)	指数、対数、平方根の関数
cplxops(3CC4)	算術演算子関数
cplxtrig(3CC4)	三角関数

第16章

ライブラリの構築

この章では、ライブラリの構築方法を説明します。

ライブラリとは

ライブラリには2つの利点があります。まず、ライブラリを使えば、コードをいくつかのアプリケーションで共有できます。共有したいコードがある場合は、そのコードを含むライブラリを作成し、コードを必要とするアプリケーションとリンクできます。次に、ライブラリを使えば、非常に大きなアプリケーションの複雑さを軽減できます。アプリケーションの中の、比較的独立した部分をライブラリとして構築および保守することで、プログラマは他の部分の作業により専念できるようになるためです。

ライブラリの構築とは、.o ファイルを作成し (コードを -c オプションでコンパイルし)、これらの .o ファイルを cc コマンドでライブラリに結合することです。ライブラリには、静的 (アーカイブ) ライブラリと動的 (共有) ライブラリがあります。

静的 (アーカイブ) ライブラリの場合は、ライブラリのオブジェクトがリンク時にプログラムの実行可能ファイルにリンクされます。アプリケーションにとって必要な .o ファイルだけがライブラリから実行可能ファイルにリンクされます。静的 (アーカイブ) ライブラリの名前には、通常、接尾辞.a が付きます。

動的 (共有) ライブラリの場合は、ライブラリのオブジェクトはプログラムの実行可能ファイルにリンクされません。その代わりに、プログラムがこのライブラリに依存することをリンカーが実行可能ファイルに記録します。プログラムが実行されるとき、システムは、プログラムに必要な動的ライブラリを読み込みます。同じ動的ライブラ

リを使用する 2 つのプログラムが同時に実行されると、ライブラリはこれらのプログラムによって共有されます。動的 (共有) ライブラリの名前には、接尾辞として .so が付きます。

共有ライブラリを動的にリンクすることは、アーカイブライブラリを静的にリンクすることに比べていくつかの利点があります。

- 実行可能ファイルのサイズが小さくなる
- 実行時にコードのかなりの部分をプログラム間で共有できるため、メモリーの使用量が少なくなる
- ライブラリを実行時に置き換える場合でも、アプリケーションとリンクし直す必要がない (プログラムの再リンクや再配布をしなくても、Solaris 環境でプログラムが新しい機能を使用できるのは、主にこの仕組みのためです)
- dlopen() 関数呼び出しを使えば、共有ライブラリを実行時に読み込むことができる

ただし、動的ライブラリには短所もあります。

- 実行時のリンクに時間がかかる
- 動的ライブラリを使用するプログラムを配布する場合には、それらのライブラリも同時に配布しなければならないことがある
- 共有ライブラリを別の場所に移動すると、システムがライブラリを検索できずに、プログラムを実行できなくなることがある (環境変数 LD_LIBRARY_PATH を使えば、この問題は解決できます)

静的 (アーカイブ) ライブラリの構築

静的 (アーカイブ) ライブラリを構築する仕組みは、実行可能ファイルを構築することに似ています。一連のオブジェクト (-.o) ファイルは、CC で -xar オプションを使うことで 1 つのライブラリに結合できます。

静的 (アーカイブ) ライブラリを構築する場合は、ar コマンドを直接使用せずに CC -xar を使用してください。C++ 言語では一般に、従来の .o ファイルに収容できる情報より多くの情報 (特に、テンプレートインスタンス) をコンパイラが持たなければなりません。-xar オプションを使用すると、テンプレートインスタンスを含め、す

べての必要な情報がライブラリに組み込まれます。make ではどのテンプレートファイルが実際に作成され、参照されているのかがわからないため、通常のプログラミング環境でこのようにすることは困難です。CC -xar を指定しないと、参照に必要なテンプレートインスタンスがライブラリに組み込まれないことがあります。例を次に示します。

```
%% CC -c foo.cc # mainを含むファイルをコンパイルし、テンプレートオブジェクトを作成する
% CC -xar -o foo.a foo.o # すべてのオブジェクトを 1 つのライブラリに集める
```

-xar フラグによって、CC が静的 (アーカイブ) ライブラリを作成します。-o 命令は、新しく作成するライブラリの名前を指定するために必要です。コンパイラは、コマンド行のオブジェクトファイルを調べ、これらのオブジェクトファイルと、テンプレートリポジトリで認識されているオブジェクトファイルとを相互参照します。そして、ユーザーのオブジェクトファイルに必要なテンプレートを (本体のオブジェクトファイルとともに) アーカイブに追加します。

注 – -xar フラグは既存のアーカイブの作成や更新のためのもので、保守には使用できません。-xar オプションはar -cr. を実行するのと同じことです。

1 つの .o ファイルには 1 つの関数を入れることをお勧めします。アーカイブとリンクする場合、特定の .o ファイルのシンボルが必要になると、.o ファイル全体がアーカイブからアプリケーションにリンクされます。.o ファイルに 1 つの関数を入れておけば、アプリケーションにとって必要なシンボルだけがアーカイブからリンクされます。

動的 (共有) ライブラリの構築

動的 (共有) ライブラリの構築方法は、コマンド行に -xar の代わりに -G を指定することを除けば、静的 (アーカイブ) ライブラリの場合と同じです。

ld は直接使用しないでください。静的ライブラリの場合と同じように、CC コマンドを使用すると、必要なすべてのテンプレートインスタンスがテンプレートリポジトリからライブラリに組み込まれます (テンプレートを使用している場合)。アプリケーションにリンクされている動的ライブラリでは、すべての静的コンストラクタは main() が実行される前に呼び出され、すべての静的デストラクタは main() が終了

した後に呼び出されます。dlopen() で共有ライブラリを開いた場合、すべての静的コンストラクタは dlopen() で実行され、すべての静的デストラクタは dlclose() で実行されます。

動的ライブラリを構築するには、必ず CC に -G を使用します。ld (リンクエディタ) または cc (C コンパイラ) を使用して動的ライブラリを構築すると、例外が機能しない場合があり、ライブラリに定義されている大域変数が初期化されません。

動的 (共有) ライブラリを構築するには、-CC の -Kpic や -KPIC オプションで各オブジェクトをコンパイルして、再配置可能なオブジェクトファイルを作成する必要があります。次に、これらの再配置可能オブジェクトファイルから動的ライブラリを構築します。原因不明のリンクエラーがいくつも出る場合は、-Kpic や -KPIC でコンパイルしていないオブジェクトがある可能性があります。

ソースファイル lsrc1.cc と lsrc2.cc から作成するオブジェクトファイルから C++ 動的ライブラリ libgoo.so.1 を構築するには、次のようにします。

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

-G オプションは動的ライブラリの構築を指定し、-o オプションはライブラリのファイル名を指定します。-h オプションは、共有ライブラリの名前を指定しています。-Kpic オプションは、オブジェクトファイルが位置に依存しないことを指定しています。

注 – CC -Gコマンドは -l オプションを ld に渡しません。共有ライブラリに他の共有ライブラリとの依存関係を持たせたい場合、必要な -l オプションをコマンド行に指定する必要があります。たとえば、共有ライブラリに libCrun.so との依存関係を持たせたい場合、-lCrun をコマンド行に指定する必要があります。

例外を含む共有ライブラリの構築

C++ コードが含まれているプログラムでは、-Bsymbolic を使用せずに、リンカーのマップファイルを使用してください。-Bsymbolic を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

`dlopen()` を使用して共有ライブラリを開いている場合、例外メカニズムが機能するには `RTLD_GLOBAL` を使用する必要があります。

非公開ライブラリの構築

ある組織の内部でしか使用しないライブラリを構築する場合には、一般的な使用には適さないオプションを使ってライブラリを構築することもできます。具体的には、ライブラリはシステムのアプリケーションバイナリインタフェース (ABI) に準拠していなくてもかまいません。たとえば、ライブラリを `-fast` オプションでコンパイルして、特定のアーキテクチャ上でのパフォーマンスを向上させることができます。同じように、`-xregs=float` オプションでコンパイルして、パフォーマンスを向上させることもできます。

公開ライブラリの構築

他の組織からも使用できるライブラリを構築する場合は、ライブラリの管理やプラットフォームの汎用性などの問題が重要になります。ライブラリを公開にするかどうかを決める簡単な基準は、アプリケーションのプログラマがライブラリを簡単に再コンパイルできるかどうかということです。公開ライブラリは、システムの ABI に準拠して構築しなければなりません。一般に、これはプロセッサ固有のオプションを使用しないということを意味します (たとえば、`-fast` や `-xtarget` は使用しないなど)。

SPARC ABI では、いくつかのレジスタがアプリケーション専用で使用されます。V7 と V8 では、これらのレジスタは `%g2`、`%g3`、`%g4` です。V9 では、これらのレジスタは `%g2` と `%g3` です。ほとんどのコンパイルはアプリケーション用に行われるので、C++ コンパイラは、デフォルトでこれらのレジスタを一時レジスタに使用して、プログラムのパフォーマンスを向上しようとします。しかし、公開ライブラリでこれらのレジスタを使用することは、SPARC ABI に適合しないことになります。公開ライブラリを構築するときには、アプリケーションレジスタを使用しないようにするために、すべてのオブジェクトを `-xregs=no%appl` オプションでコンパイルしてください。

C API を持つライブラリの構築

C++ で作成されたライブラリを C プログラムから使用できるようにするには、C API を作成する必要があります。そのためには、エクスポートされるすべての関数を `extern "C"` にします。ただし、これができるのは大域関数だけで、メンバー関数にはできません。

C インタフェースライブラリで C++ の実行時サポートを必要とし、しかも cc とリンクしている場合は、C インタフェースライブラリを使用するときにアプリケーションも `libC` (互換モード) または `libCrun` (標準モード) にリンクする必要があります (C インタフェースライブラリで C++ 実行時サポートが不要の場合は、`libC` や `libCrun` とリンクする必要はありません) 。リンク手順は、アーカイブされたライブラリと共有ライブラリでは異なります。

アーカイブされた C インタフェースライブラリを提供するときは、ライブラリの使用方法を説明する必要があります。

- C インタフェースライブラリが `CC` を標準モード (デフォルト) で構築している場合は、C インタフェースライブラリを使用するときに `-lCrun` を `cc` コマンド行に追加します。
- C インタフェースライブラリが `CC` を互換モード (`-compat`) で構築している場合は、C インタフェースライブラリを使用するときに `-lC` を `cc` コマンド行に追加します。

共有 C インタフェースライブラリを提供するときは、ライブラリの構築時に `libC` または `libCrun` と依存関係をつくる必要があります。共有ライブラリの依存関係が正しければ、ライブラリを使用するときに `-lC` または `-lCrun` をコマンド行に追加する必要はありません。

- C インタフェースライブラリを互換モード (`-compat`) で構築している場合は、`-lC` インタフェースライブラリを使用するときに `-lC` を `cc` コマンド行に追加します。ライブラリ構築時に`-lC` を`CC` コマンド行に追加します。
- C インタフェースライブラリを標準モード (デフォルト) で構築している場合は、C インタフェースライブラリを使用するときに `-lCrun` を `cc`ではなく`CC`コマンド行に追加します。

さらに、C++ 実行時ライブラリにもまったく依存しないようにするには、ライブラリソースに対して次のコーディング規則を適用する必要があります。

- どのような形式の new も delete も使用しない (独自の new または delete を定義する場合は除く)
- 例外を使用しない
- 実行時の型識別機構 (RunTime Type Information、RTTI) を使用しない

dlopen を使って C プログラムから C++ ライブラリにアクセスする

C プログラムから dlopen で C++ 共有ライブラリを開く場合は、共有ライブラリが適切な C++ 実行時ライブラリ (-compat=4 の場合は libC.so.5、-compat=5 の場合は libCrun.so.1) に依存していなければなりません。

そのためには、共有ライブラリを構築するときに、-compat=4 の場合は -lc-lC、-compat=5 の場合は -lCrun を次のようにコマンド行に追加します。

```
example% CC -G -compat=4 ... -lC
example% CC -G -compat=5 ... -lCrun
```

共有ライブラリが例外を使用している場合には、ライブラリが C++ 共有ライブラリに依存していないと、C プログラムが正しく動作しないことがあります。

注 – 共有ライブラリを dlopen() で開く場合は、RTLD_GLOBAL を使用しないと、例外は機能しません。

PART IV　付録

付録 A

C++ コンパイラオプション

この付録では、Solaris 7 と Solaris 8 のオペレーティング環境で稼働する CC コンパイラのコマンド行オプションについて詳しく説明します。これらの機能は、特に記載がない限りすべてのプラットフォームに適用されます。Solaris SPARC プラットフォーム版のオペレーティング環境に特有の機能は SPARC として表記され、Solaris Intel プラットフォーム版のオペレーティング環境に特有の機能は IA として表記されます。

次の表には従来のオプション構文形式の例を示します。

表 A-1 オプション構文形式の例

構文形式	例
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

この節では、個別のオプションを説明するために、このマニュアルの先頭にある「はじめに」に記載した表記上の規則を使用しています。

括弧、中括弧、角括弧、パイプ文字、および省略符号は、オプションの説明で使用されているメタキャラクタです。これらは、オプションの一部ではありません。

オプション情報の構成

簡単に情報を検索できるように、次の見出しに分けてコンパイラオプションを説明しています。オプションが他のオプションで置き換えられたり、他のオプションと同じである場合、詳細については他のオプション説明を参照してください。

表 A-2 オプションの見出し

見出し	意味
オプションの定義	各オプションのすぐ後には短い定義があります (小見出しはありません)。
値	オプションに値がある場合は、その値を示します。
デフォルト	オプションに一次または二次のデフォルト値がある場合は、それを示します。 一次のデフォルトとは、オプションが指定されなかったときに有効になるオプションの値です。たとえば、`-compat` を指定しないと、デフォルトは `-compat=5` になります。 二次のデフォルトとは、オプションは指定されたが、値が指定されなかったときに有効になるオプションの値です。たとえば、値を指定せずに `-compat` を指定すると、デフォルトは `-compat=4` になります。
拡張	オプションにマクロ展開がある場合は、ここに示します。
例	オプションの説明のために例が必要な場合は、ここに示します。
相互の関連性	他のオプションとの相互の関連性がある場合は、その関係をここに示します。

表 A-2　オプションの見出し (続き)

見出し	意味
警告	オプションの使用について注意がある場合はここに示します。予測できない動作の原因となる操作についてもここに示します。
関連項目	ここには、参考情報が得られる他のオプションや文書を示します。
置き換え、同じなどの言葉	そのオプションが廃止され、他のもので置き換えられていたり、そのオプションの代わりに別のオプションを使用する方がよい場合は、置き換えるオプションを「置き換え」や「同じ」という表記とともに示しています。このような指示のあるオプションは、将来のリリースでサポートされない可能性があります。 一般的な意味と目的が同じであるオプションが 2 つある場合は、望ましいオプションを示します。たとえば、「-xO と同じです」は、-xO が望ましいオプションであることを示します。

オプションの一覧

-386

IA:-xtarget=386 と同じです。このオプションは、下位互換のためだけに用意されています。

-486

IA:-xtarget=486 と同じです。このオプションは、下位互換のためだけに用意されています。

-a

-xa と同じです。

-B*binding*

ライブラリのリンク形式を、シンボリックか、動的 (共有ライブラリ) にするか、静的 (共有でないライブラリ) のいずれかからを指定します。

-B オプションは同じコマンド行で何回も指定することができます。このオプションはリンカー (ld) に渡されます。

注 – Solaris 7 および Solaris 8 プラットフォームでは、必ずしもすべてのライブラリが静的ライブラリとして使用できるわけではありません。

値

binding には次のいずれかの値を指定します。

binding の値	意味
dynamic	まずlib*lib*.so (共有) ファイルを検索するようにリンカーに指示します。これらのファイルが見つからないと、リンカーは lib*lib*.a (静的で、共有されない) ファイルを検索します。ライブラリのリンク方式を共有にしたい場合は、このオプションを指定します。
static	-Bstatic オプションを指定すると、リンカーは lib*lib*.a (静的で、共有されない) ファイルだけを検索します。ライブラリのリンク形式を非共有にしたい場合は、このオプションを指定します。
symbolic	シンボルがほかですでに定義されている場合でも、可能であれば共有ライブラリ内でシンボル解決を実行します。ld(1) のマニュアルページを参照してください。

-B と *binding* との間に空白があってはなりません。

デフォルト

-B を指定しないと、-Bdynamic が使用されます。

相互の関連性

C++ のデフォルトのライブラリを静的にリンクするには、-staticlib オプションを使用します。

-Bstatic および -Bdynamic オプションは、デフォルトで使用されるライブラリのリンクにも影響します。デフォルトのライブラリを動的にリンクするには、最後に指定する -B が -Bdynamic でなければなりません。

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには libm.so および libc.so があります (libm.a と libc.a は提供していません)。その結果、-Bstatic と -dn を使用すると 64 ビットの Solaris オペレーティング環境でリンクエラーが生じる可能性があります。この場合、アプリケーションを動的ライブラリとリンクさせる必要があります。

例

次の例では、lib*foo*.so があっても lib*foo*.a がリンクされます。他のすべてのライブラリは動的にリンクされます。

```
example% CC a.o -Bstatic -lfoo -Bdynamic
```

警告

C++ コードが含まれているプログラムでは、-Bsymbolic を使用せずに、リンカーのマップファイルを使用してください。

-Bsymbolic を使用すると、異なるモジュール内の参照が、本来 1 つの大域オブジェクトの複数の異なる複製に結合されてしまう可能性があります。

例外メカニズムは、アドレスの比較によって機能します。オブジェクトの複製が 2 つある場合は、アドレスが同一であると評価されず、本来一意のアドレスを比較することで機能する例外メカニズムで問題が発生することがあります。

コンパイルとリンクを別々に行う場合で、コンパイル時に *-Bbinding* オプションを使用した場合は、このオプションをリンク時にも指定する必要があります。

関連項目

-nolib, -staticlib, ld(1)、166 ページの「標準ライブラリの静的リンク」、『リンカーとライブラリ』

-c

コンパイルのみ。オブジェクト .o ファイルを作成しますが、リンクはしません。

この オプションは ld によるリンクを抑止し、各ソースファイルに対する .o ファイルを 1 つずつ生成するように、CC ドライバに指示します。コマンド行にソースファイルを 1 つだけ指定する場合には、-o オプションでそのオブジェクトファイルに明示的に名前を付けることができます。

例

CC -c x.cc と入力すると、x.o というオブジェクトファイルが生成されます。

CC -c x.cc -o y.o と入力すると、y.o というオブジェクトファイルが生成されます。

警告

コンパイラは、入力ファイル (.c、.i) に対するオブジェクトコードを作成する際に、.o ファイルを作業ディレクトリに作成します。リンク手順を省略すると、この .o ファイルは削除されません。

関連項目

-o *filename*, -xe

-cg{89|92}

-xcg{89|92} と同じです。

-compat[={4|5}]

コンパイラの主要リリースとの互換モードを設定します。このオプションは、__SUNPRO_CC_COMPAT と __cplusplus マクロを制御します。

C++ コンパイラには主要なモードが 2 つあります。1 つは互換モードで、4.2 コンパイラで定義された ARM の意味解釈と言語が有効です。もう1 つは標準モードです。このモードでは、構文は ANSI/ISO 標準に従っていなければなりません。これらのモードには互換性はありません。ANSI/ISO 標準では、名前の符号化、vtable の配置、その他の ABI の細かい点で互換性のない変更がかなり必要であるためです。これらのモードは、次に示す -compat オプションで指定します。

値

-compat オプションには次の値を指定できます。

値	意味
-compat=4	(互換モード) 言語とバイナリの互換性を 4.0.1、4.1、4.2 コンパイラに合わせます。__cplusplus プリプロセッサマクロを 1 に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 4 にそれぞれ設定します。
-compat=5	(標準モード) 言語とバイナリの互換性を ANSI/ISO 標準モード 5.0 コンパイラに合わせます。__cplusplus プリプロセッサマクロを 1997IIL に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 5 にそれぞれ設定します。

デフォルト

-compat オプションを指定しないと、-compat=5 が使用されます。

-compat だけを指定すると、-compat=4 が使用されます。

_SUNPRO_CC は、-compat の設定に関係なく 0x550 に設定されます。

相互の関連性

標準ライブラリは互換モード (-compat[=4]) で使用できません。

-compat[=4] では次のオプションの使用はサポートしていません。

- ライブラリに例外がある場合は -Bsymbolic
- features=[no%]strictdestrorder
- features=[no%]tmplife
- library=[no%]iostream

- library=[no%]Cstd
- library=[no%]Crun
- library=[no%]rwtools7_std
- xarch=native64、-xarch=generic64、-xarch=v9、-xarch=v9a、または -xarch=v9b

-compat=5 では次のオプションの使用はサポートされません。

- +e
- features=[no%]arraynew
- features=[no%]explicit
- features=[no%]namespace
- features=[no%]rtti
- library=[no%]complex
- library=[no%]libC
- -vdelx

警告

共有ライブラリを互換モード (-compat[=4]) で構築するときに、ライブラリに例外がある場合は -B*symbolic* を使用しないでください。獲得する必要のある例外を逃す可能性があります。

関連項目

『**C++** 移行ガイド』

+d

C++ インライン関数を展開しません。

C++ 言語の規則では、C++ は、次の条件のうち 1 つがあてはまる場合にインライン化します。

- 関数が inline キーワードを使用して定義されている
- 関数がクラス定義の中に (宣言されているだけでなく) 定義されている
- 関数がコンパイラで生成されたクラスメンバー関数である

C++ 言語の規則では、呼び出しを実際にインライン化するかどうかをコンパイラが選択します。ただし、次の場合を除きます。

- 関数が複雑すぎる、
- +d オプションが選択されている、または
- -g オプションが選択されている

例

デフォルトでは、コンパイラは次のコード例で関数 f() と memf2() をインライン化できます。また、クラスには、コンパイラによって生成されたデフォルトのコンストラクタとコンパイラでインライン化できるデストラクタがあります。+d を使用すると、コンパイラでコンストラクタ f() とデストラクタ C::mf2() はインライン化されません。

```
inline int f() { return 0; } //おそらくインライン化される
class C {
  int mf1(); // インライン定義が出現するまではインライン化されない
  int mf2() { return 0; } // おそらくインライン化される
};
```

相互の関連性

デバッグオプション -g を指定すると、このオプションが自動的に有効になります。

-g0 デバッグオプションでは、+d は有効になりません。

+d オプションは、-x04 または -x05 を使用するときに実行される自動インライン化に影響を与えません。

関連項目

-g0, -g

-D[ ]*name*[=*def*]

プリプロセッサに対してマクロシンボル名 *name* を def と定義します。

このオプションは、ソースファイルの先頭に #define 指令を記述するのと同じです。-D オプションは複数指定できます。

値

次の表は、事前に定義されているマクロを示しています。これらの値は、#ifdef のようなプリプロセッサに対する条件式の中で使用できます。

表 A-3　SPARC と IA 用の事前定義シンボル

型	マクロ名	注
SPARC と IA	__ARRAYNEW	「配列」形式の演算子 new と delete を有効にしてコンパイルした場合に使用される。詳細は -features=[no%]arraynew を参照。
	_BOOL	ブール型を有効にした場合に使用される。詳細は -features=[no%]bool を参照。
	__BUILTIN_VA_ARG_INCR	varargs.h、stdarg.h、sys/varags.h のキーワードが __builtin_alloca、__builtin_va_alist、__builtin_va_arg_incr の場合に使用される。
	__cplusplus	
	__DATE__	
	__FILE__	
	__LINE__	
	__STDC__	
	__sun	
	sun	「相互の関連性」を参照。
	__SUNPRO_CC=0x550	__SUNPRO_CC の値はコンパイラのリリース番号を表す。
	___SUNPRO_CC_COMPAT=4 または __SUNPRO_CC_COMPAT=5	250 ページの「-compat[=4|5]」参照。
	__SVR4	
	__TIME__	

表 A-3　**SPARC** と **IA** 用の事前定義シンボル **(**続き**)**

型	マクロ名	注
	__'*uname -s*'_'*uname -r*'	*uname -s* は uname -s の出力で、*uname -r* は uname -r の出力。無効な文字 (ピリオドなど) は下線で置き換えられる (例: -D__SunOS_5_7、-D__SunOS_5_8) 。
	__unix	
	unix	「相互の関連性」を参照。
SPARC	__sparc	
	sparc	「相互の関連性」を参照。
SPARC v9	__sparcv9	64 ビットコンパイルモードのみ
IA	__i386	
	i386	「相互の関連性」を参照。
UNIX	_WCHAR_T	

=*def* を使用しないと、*name* は 1 になります。

相互の関連性

+p が使用されている場合は、sun、unix、sparc、i386 は定義されません。

関連項目

-U

-d{y|n}

実行可能ファイル全体に対して動的ライブラリを使用できるかどうか指定します。

このオプションは ld に渡されます。

このオプションは、コマンド行では 1 度だけしか使用できません。

値

値	意味
-dy	リンカーで動的リンクを実行します。
-dn	リンカーで静的リンクを実行します。

デフォルト

-d オプションを指定しないと、-dy が使用されます。

相互の関連性

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには libm.so および libc.so があります (libm.a と libc.a は提供していません)。その結果、-Bstatic と -dn を使用すると 64 ビットの Solaris オペレーティング環境でリンクエラーが生じる可能性があります。この場合、アプリケーションを動的ライブラリとリンクさせる必要があります。

関連項目

ld(1)、『リンカーとライブラリ』

-dalign

-dalign は、-xmemalign=8s を指定することと同じです。詳細については、369 ページの「-xmemalign=ab」を参照してください。

警告

あるプログラム単位を -dalign でコンパイルした場合は、プログラムのすべての単位を -dalign でコンパイルしなければなりません。そうしないと予期しない結果が生じることがあります。

-dryrun

ドライバによって作成されたコマンドを表示しますが、コンパイルはしません。

このオプションは、コンパイルドライバが作成したサブコマンドの表示のみを行い、実行はしないように CC ドライバ に指示します。

-E

ソースファイルに対してプリプロセッサを実行しますが、コンパイルはしません。

C++ のソースファイルに対してプリプロセッサだけを実行し、結果を stdout (標準出力) に出力するよう CC ドライバに指示します。コンパイルは行われません。したがって .o ファイルは生成されません。

このオプションを使用すると、プリプロセッサで作成されるような行番号情報が出力に含まれます。

例

このオプションは、プルプロセッサの処理結果を知りたいときに便利です。たとえば、コード例 A-2のようなプログラム foo.cc があるとします。

コード例 A-1 プルプロセッサのプログラム例 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
```

コード例 A-2 -E オプションを使用したときの foo.cc のプリプロセッサ出力

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power ( int , int ) ;


int main ( ) {
int x ;
x = power ( 2 , 10 ) ;
}
```

警告

テンプレートを使用する場合は、このオプションの結果を C++ コンパイラの入力に使用することはできません。

関連項目

-P

+e{0|1}

互換モード (-compat=4) のときに仮想テーブルの生成を制御します。標準モード (デフォルトモード) のときには無効な指定として無視されます。

値

+e オプションには次の値を指定できます。

値	意味
0	仮想テーブルを生成せず、必要とされているテーブルへの外部参照を生成します。
1	仮想関数を使用して定義したすべてのクラスごとに仮想テーブルを生成します。

相互の関連性

このオプションを使用してコンパイルする場合は、`-features=no%except` オプションも使用してください。使用しなかった場合は、例外処理で使用される内部型の仮想テーブルがコンパイラによって生成されます。

テンプレートクラスに仮想関数があると、コンパイラで必要な仮想テーブルがすべて生成され、しかもこれらのテーブルが複写されないようにすることができない場合があります。

関連項目

『**C++** 移行ガイド』

-erroff[=*t*]

このコマンドは、C++ コンパイラの警告メッセージを無効にします。エラーメッセージには影響しません。

値

t には、以下の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、`%all`、`%none`。指定順序によって実行内容が異なります。たとえば、「`%all,no%`*tag*」と指定すると、*tag* 以外のすべての警告メッセージを抑制します。次の表は、`-erroff` の値を示しています。

表 A-4 `-erroff` の値

値	意味
tag	*tag* のリストに指定されているメッセージを抑制します。`-errtags=yes` オプションを使用すればメッセージのタグを表示することができます。
`no%` *tag*	*tag* 以外のすべての警告メッセージの抑制を解除します。
`%all`	すべての警告メッセージを抑制します。
`%none`	すべてのメッセージの抑制を解除します (デフォルト)。

デフォルト

デフォルトは -erroff=%none です。-erroff と指定すると、-erroff=%all を指定した場合と同じ結果が得られます。

例

たとえば、-erroff=*tag* は、この *tag* が示す警告メッセージを抑止します。一方、-erroff=%all,no%*tag* は、*tag* が示すメッセージ以外の警告メッセージをすべて抑止します。

警告メッセージのタグを表示するには、-errtags=yes オプションを使用します。

警告

-erroff オプションで無効にできるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

関連項目

-errtags, -errwarn

-errtags[=*a*]

C++ コンパイラのフロントエンドで出力される警告メッセージのうち、-erroff オプションで無効にできる、または -errwarn オプションで重大な警告に変換できるメッセージのメッセージタグを表示します。

値とデフォルト

a には、yes または no を指定します。デフォルトは -errtags=no です。-errtags だけを指定すると、-errtags=yes を指定するのと同じことになります。

警告

C++ コンパイラのドライバおよび C のコンパイルシステムの他のコンポーネントから出力されるメッセージにはエラータグが含まれないため、-erroff で無効にしたり、-errwarn で重大なエラーに変換したりすることはできません。

関連項目

-erroff、-errwarn

-errwarn[=*t*]

指定した警告メッセージが生成された場合に、重大なエラーを出力して C++ コンパイラを終了する場合は、-errwarn を使用します。

値

t には、以下の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。このとき、順序が重要になります。たとえば、%all,no%*tag* と指定すると、*tag* 以外のすべての警告メッセージが生成された場合に、重大なエラーを出力して cc を終了します。

-errwarn の値を次の表に示します。

表 A-5　-errwarn の値

値	意味
tag	*tag* に指定されたメッセージが警告メッセージとして発行されると、cc は致命的エラーステータスを返して終了します。*tag* に指定されたメッセージが発行されない場合は無効です。
no%*tag*	*tag* に指定されたメッセージが警告メッセージとしてのみ発行された場合に、cc が致命的なエラーステータスを返して終了しないようにします。*tag* に指定されたメッセージが発行されない場合は無効です。このオプションは、*tag* または %all を使用して以前に指定したメッセージが警告メッセージとして発行されても cc が致命的エラーステータスで終了しないようにする場合に使用してください。
%all	警告メッセージが 1 つでも発行されると cc は致命的ステータスを返して終了します。%all に続いて no%*tag* を使用して、特定の警告メッセージを対象から除外することもできます。
%none	どの警告メッセージが発行されても cc が致命的エラーステータスを返して終了することがないようにします。

デフォルト

デフォルトは、-errwarn=%none です。-errwarn だけを指定した場合、-errwarn=%all と指定したことと同じになります。

警告

-errwarn オプションを使用して、障害状態で C++ コンパイラを終了するように指定できるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

C++ コンパイラで生成される警告メッセージは、コンパイラのエラーチェックの改善や機能追加に応じて、リリースごとに変更されます。-errwarn=%all を指定してエラーなしでコンパイルされるコードでも、コンパイラの次期リリースではエラーを出力してコンパイルされる可能性があります。

関連項目

-erroff、-errtags

-fast

コンパイルオプションの最適な組み合わせを選択し、実行速度を最適化します。

このオプションは、コードをコンパイルするマシン上でコンパイラオプションの最適な組み合わせを選択して実行速度を向上するマクロです。

拡張

このオプションは、次のコンパイラオプションを組み合わせて、多くのアプリケーションのパフォーマンスをほぼ最大にします。

表 A-6　-fast の拡張

オプション	SPARC	IA
-fns	○	○
-fsimple=2	○	—
-ftrap=%none	○	○
-nofstore	—	○

表 A-6　-fast の拡張 (続き)

オプション	SPARC	IA
-xarch	○	○
-xlibmil	○	○
-xlibmopt	○	○
-xmemalign	○	
-xO5	○	○
-xtarget=native	○	○
-xbuiltin=%all	○	○

相互の関連性

-fast マクロから展開されるコンパイラオプションが、指定された他のオプションに影響を与えることがあります。たとえば、次のコマンドの -fast マクロの展開には -xtarget=native が含まれています。そのため、ターゲットのアーキテクチャは -xarch に指定された SPARC-V9 ではなく、32 ビットアーキテクチャのものに戻されます。

誤

```
example% CC -xarch=v9 -fast test.cc
```

正

```
example% CC -fast -xarch=v9 test.cc
```

個々の相互の関連性については、各オプションの説明を参照してください。

このコード生成オプション、最適化レベル、組み込み関数の最適化、インラインテンプレートファイルの使用よりも、その後で指定するフラグの方が優先されます(例を参照)。ユーザーの指定した最適化レベルは、以前に設定された最適化レベルを無効にします。

-fast オプションには -fns -ftrap=%none が含まれているため、このオプションによってすべてのトラップが無効になります。

例

次のコンパイラコマンドでは、最適化レベルは -xO3 になります。

```
example% CC -fast -xO3
```

次のコンパイラコマンドでは、最適化レベルは -xO5 になります。

```
example% CC -xO3 -fast
```

警告

別々の手順でコンパイルしてリンクする場合は、-fast オプションをコンパイルコマンドとリンクコマンドの両方に表示する必要があります。

コンパイラで -fast オプションを指定すると、そのコードの移植性は失われます。たとえば、UltraSPARC-III システムで次のコマンドを指定すると、生成されるバイナリは UltraSPARC-II システムでは動作しません。

```
example% CC -fast test.cc
```

IEEE 標準の浮動小数点演算を使用しているプログラムには、-fast を指定しないでください。計算結果が違ったり、プログラムが途中で終了する、あるいは予期しない SIGFPE シグナルが発生する可能性があります。

以前のリリースの SPARC では、-fast マクロは -fsimple=1 に展開されました。現在では、-fsimple=2 に展開されます。

以前のリリースでは、-fast マクロは -xO4 に展開されました。現在では、-xO5 に展開されます。

注 – 以前の SPARC リリースでは -fast マクロに -fnonstd が含まれていましたが、このリリースでは含まれていません。-fast では、非標準浮動小数点モードは初期化されません。『数値計算ガイド』と ieee_sun(3M) のマニュアルページを参照してください。

関連項目

-fns、-fsimple、-ftrap=%none、-xlibmil、-nofstore、-xO5、-xlibmopt、-xtarget=native

-features=*a*[,*a*...]

コンマで区切って指定された C++言語のさまざまな機能を、有効または無効にします。

値

互換モード (-compat[=4]) と標準モード (デフォルトのモード) の両方で、次の値の 1 つを指定できます。

表 A-7 互換モードと標準モードでの -feature オプション

a の値	意味
%all	指定されているモードに対して有効なすべての -feature オプションを有効にします。
[no%]altspell	トークンの代替スペル(たとえば、&& の代わりの and) を認識します [しません]。デフォルトは互換モードで no%altspell、標準モードで altspell です。
[no%]anachronisms	廃止されている構文を許可します [しません]。無効にした場合 (つまり、-features=no%anachronisms)、廃止されている構文は許可されません。デフォルトは anachronisms です。
[no%]bool	ブール型とリテラルを許可します [しません]。有効にした場合、マクロ _BOOL=1 が定義されます。有効にしないと、マクロは定義されません。デフォルトは互換モードで no%bool、標準モードで bool です。
[no%]conststrings	リテラル文字列を読み取り専用メモリーに入れます [入れません]。デフォルトは互換モードで no%conststrings、標準モードで conststrings です。

表 A-7　互換モードと標準モードでの -feature オプション (続き)

a の値	意味
[no%]except	C++ 例外を許可します [しません]。C++ 例外を無効にした場合 (つまり、-features=no%except)、関数に指定された throw は受け入れられますが無視されます。つまり、コンパイラは例外コードを生成しません。キーワード try、throw、および catch は常に予約されています。104 ページの「例外の無効化」を参照してください。デフォルトは except です。
[no%]export	キーワード export を認識します [しません]。デフォルトは互換モードで no%export、標準モードで export です。
[no%]extensions	他の C++ コンパイラによって一般に受け入れられた非標準コードを許可します [しません]。-features=extensions オプションを使用するときにコンパイラによって受け入れられる無効なコードの説明については第 4 章を参照してください。デフォルトは no%extensions です。
[no%]iddollar	識別子の最初以外の文字に $ を許可します [しません]。デフォルトは no%iddollar です。
[no%]localfor	for 文に対して新しい局所スコープ規則を使用します [しません]。デフォルトは互換モードで no%localfor、標準モードで localfor です。
[no%]mutable	キーワード mutable を認識します [しません]。デフォルトは互換モードで no%mutable、標準モードで mutable です。

表 A-7　互換モードと標準モードでの -feature オプション (続き)

a の値	意味
[no%]split_init	非ローカル静的オブジェクトの初期設定子を個別の関数に入れます [入れません]。-feature=no%split_init を使用すると、コンパイラではすべての初期設定子が 1 つの関数に入れられます。-features=no%split_init を使用すると、コンパイル時間を可能な限り費やしてコードサイズを最小化します。デフォルトは split_init です。
[no%]transitions	標準 C++ で問題があり、しかもプログラムが予想とは違った動作をする可能性があるか、または将来のコンパイラで拒否される可能性のある ARM 言語構造を許可します [しません]。-feature=no%transitions を使用すると、コンパイラではこれらの言語構造をエラーとして扱います。 -feature=transitions を標準モードで使用すると、これらの言語構造に関してエラーメッセージではなく警告が出されます。-features=transitions を互換モード (-compat[=4]) で使用すると、コンパイラでは +w または +w2 が指定された場合に限りこれらの言語構造に関する警告が表示されます。次の構造は移行エラーとみなされます。テンプレートの使用後にテンプレートを再定義する、typename 指示をテンプレートの定義に必要なときに省略する、int 型を暗黙的に宣言する。一連の移行エラーは将来のリリースで変更される可能性があります。デフォルトは transitions です。
%none	指定されているモードに対して無効にできるすべての機能を無効にします。

標準モード (デフォルトのモード) では、*a* にはさらに次の値の 1 つを指定できます。

表 A-8 標準モードだけに使用できる `-features` オプション

a の値	意味
`[no%]strictdestrorder`	静的記憶領域にあるオブジェクトを破棄する順序に関する、C++ 標準の必要条件に従います[従いません]。デフォルトは `strictdestrorder` です。
`[no%]tmplife`	完全な式の終わりに式によって作成される一時オブジェクトを ANSI/ISO C++ 標準の定義に従って整理します [しません] (`-features=no%tmplife` が有効である場合は、大多数の一時オブジェクトはそのブロックの終わりに整理されます)。デフォルトは`no%tmplife` です。

互換モード (`-compat[=4]`) では、*a* にはさらに次の値の 1 つを指定できます。

表 A-9 互換モードだけに使用できる `-features` オプション

a の値	意味
`[no%]arraynew`	`operator new` と `operator delete` の配列形式を認識します [しません] (たとえば、`operator new [ ] (void*)`)。これを有効にすると、マクロ `__ARRAYNEW=1`.が定義されます。有効にしないと、マクロは定義されません。デフォルトは `no%arraynew` です。

表 A-9　互換モードだけに使用できる -features オプション (続き)

a の値	意味
[no%]explicit	キーワード explicit を認識します [しません]。デフォルトは no%explicit です。
[no%]namespace	キーワード namespace と using を許可します [しません]。デフォルトは no%namespace です。 -features=namespace は、コードを標準モードに変換しやすくするために使用します。このオプションを有効にすると、これらのキーワードを識別子として使用している場合にエラーメッセージが表示されます。キーワード認識オプションを使用すると、標準モードでコンパイルすることなく、追加キーワードが使用されているコードを特定することができます。
[no%]rtti	実行時の型識別 (RTTI) を許可します [しません]。dynamic_cast<> および typeid演算子を使用する場合は、RTTI を有効にする必要があります。デフォルトは no%rtti です。

注 – [no%]castop は、C++ 4.2 コンパイラ用に作成された makefile との互換性を維持するために使用できますが、C++ 5.0、5.1、5.2、および 5.3 コンパイラには影響はありません。新しい書式の型変換 (const_cast、dynamic_cast、reinterpret_cast、static_cast) は常に認識され、無効にすることはできません。

デフォルト

-features を指定しないと、以下が使用されます。

- 互換モード (-compat[=4])

```
-features=%none,anachronisms,except,split_init,transitions
```

- 標準モード (デフォルトモード)

```
-features=%all,no%iddollar,no%extensions
```

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

次の値の標準モードによる使用 (デフォルト) は、標準ライブラリやヘッダと互換性がありません。

- `no%bool`
- `no%except`
- `no%mutable`
- `no%explicit`

互換モード (`-compat[=4]`) では、`+w` オプションまたは `+w2` オプションを指定しない限り、`-features=transitions` オプションは無効です。

警告

`-features=tmplife` オプションを使用すると、プログラムの動作が変わる場合があります。プログラムが `-features=tmplife` オプションを指定してもしなくても動作するかどうかをテストする方法は、プログラムの移植性をテストする方法の 1 つです。

コンパイラはデフォルトで `-features=split_init` をとります。`-features=%none` オプションを使用して他の機能を使用できないようにした場合は、代わりに `-features=%none,split_init` を使用して初期設定子の個別の関数への分割をまた有効にすることをお勧めします。

関連項目

第 4 章および『C++ 移行ガイド』

`-filt`[=*filter*[,*filter*...]]

コンパイラによってリンカーとコンパイラのエラーメッセージに通常適用されるフィルタリングを制御します。

値

filter は次の値のいずれである必要があります。

表 A-10 `-filt` の値

filter の値	意味
[no%]errors	C++ のリンカーエラーメッセージの説明を表示します [しません]。説明の抑止は、リンカーの診断を別のツールに直接提供している場合に便利です。
[no%]names	C++ で符号化されたリンカー名を復号化します [しません]。
[no%]returns	関数の戻り型を復号化します [しません]。この種の復号化を抑止すると、より迅速に関数名が識別しやすくなりますが、共有の不変式の戻り値の場合、一部の関数は戻り型でのみ異なることに注意してください。
[no%]stdlib	リンカーとコンパイラの両方のエラーメッセージに出力される標準ライブラリからの名前を簡略化します。この結果、標準ライブラリテンプレート型の名前を容易に認識できるようになります。
%all	`-filt=errors,names,returns,stdlib` に相当します。これはデフォルトの動作です。
%none	`-filt=no%errors,no%names,no%returns,no%stdlib` に相当します。

デフォルト

`-filt` オプションを指定しないで、または値を入れないで `-filt` を指定すると、コンパイラでは`-filt=%all` が使用されます。

例

次の例では、このコードを -filt オプションでコンパイルしたときの影響を示します。

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // 定義なし
};

int main ()
{
    type t;
}
```

-filt オプションを指定しないでコードをコンパイルすると、コンパイラでは -filt=errors,names,returns,stdlib が使用され、標準出力が表示されます。

```
example% CC filt_demo.cc
Undefined              first referenced
 symbol                    in file
type::~type()          filt_demo.o
type::__vtbl           filt_demo.o
[Hint: try checking whether the first non-inlined, non-pure
virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコマンドでは、C++ で符号化されたリンカー名の復号化が抑止され、C++ のリンカーエラーの説明が抑止されます。

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコードについて考えてみましょう。

```
#include <string>
#include <list>
int main ()
{
    std::list<int> l;
    std::string s(l); // error here
}
```

以下は、-filt=no%stdlib を指定したときの出力です。

```
Error: Cannot use std::list<int, std::allocator<int>> to
initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

以下は、-filt=stdlib を指定したときの出力です。

```
Error: Cannot use std::list<int> to initialize std::string .
```

相互の関連性

no%names を使用しても returns や no%returns に影響はありません。つまり、次のオプションは同じ効果を持ちます。

- -filt=no%names
- -filt=no%names,no%returns
- -filt=no%names,returns

-flags

-xhelp=flags と同じです。

-fnonstd

浮動小数点オーバーフローのハードウェアによるトラップ、ゼロによる除算、無効演算の例外を有効にします。これらの結果は、SIGFPE シグナルに変換されます。プログラムに SIGFPE ハンドラがない場合は、メモリーダンプを行ってプログラムを終了します (ただし、コアダンプのサイズをゼロに制限した場合を除きます)。

SPARC:さらに、-fnonstd は SPARC 非標準浮動小数点を選択します。

デフォルト

-fnonstd を指定しないと、IEEE 754 浮動小数点演算例外が起きても、プログラムは異常終了しません。アンダーフローは段階的です。

拡張

IA:-fnonstd は -ftrap=common に拡張されます。

SPARC:-fnonstd は -fns -ftrap=common に拡張されます。

関連項目

-fns、-ftrap=common、『数値計算ガイド』

-fns[={yes|no}]

SPARC:SPARC 非標準浮動小数点モードを有効または無効にします。

-fns=yes (または -fns) を指定すると、プログラムが実行を開始するときに、非標準浮動小数点モードが有効になります。

このオプションを使うと、-fns を含む他のマクロオプション (-fast など) の後で非標準と標準の浮動小数点モードを切り替えることができます (「例」を参照)。

一部の SPARC デバイスでは、非標準浮動小数点モードで「段階的アンダーフロー」が無効にされ、非正規の数値を生成する代わりに、小さい値がゼロにフラッシュされます。さらに、このモードでは、非正規のオペランドが報告なしにゼロに置き換えられます。

段階的アンダーフローや、非正規の数値をハードウェアでサポートしない SPARC デバイスでは、-fns=yes (または -fns) を使用すると、プログラムによってはパフォーマンスが著しく向上することがあります。

値

-fns オプションには次の値を指定できます。

表 A-11 -fns の値

値	意味
yes	非標準浮動小数点モードを選択します。
no	標準浮動小数点モードを選択します。

デフォルト

-fns を指定しないと、非標準浮動小数点モードは自動的には有効にされません。標準の IEEE 754 浮動小数点計算が行われます。つまり、アンダーフローは段階的です。

-fns だけを指定すると、-fns=yes とみなされます。

例

次の例では、-fast は複数のオプションに展開され、その中には -fns=yes (非標準浮動小数点モードを選択する) も含まれます。ところが、その後に続く -fns=no が初期設定を変更するので、結果的には、標準の浮動小数点モードが使用されます。

```
example% CC foo.cc -fast -fns=no
```

警告

非標準モードが有効になっていると、浮動小数点演算によって、IEEE 754 規格の条件に合わない結果が出力されることがあります。

1 つのルーチンを -fns でコンパイルした場合は、そのプログラムのすべてのルーチンを-fns オプションでコンパイルする必要があります。そうしないと、予期しない結果が生じることがあります。

このオプションは、SPARC プラットフォームでメインプログラムをコンパイルするときしか有効ではありません。IA プラットフォームでは、このオプションは無視されます。

-fns=yes (または-fns のみ) を使用したときに、通常は IEEE 浮動小数点トラップハンドラによって管理される浮動小数点エラーが発生すると、次のメッセージが返されることがあります。

関連項目

『数値計算ガイド』、ieee_sun(3M)

-fprecision=*p*

(IA) デフォルト以外の浮動小数点精度モードを設定します。

-fprecision オプションを指定すると、FPU (Floating Point Unit) 制御ワードの丸め精度モードのビットが設定されます。これらのビットは、基本演算 (加算、減算、乗算、除算、平方根) の結果をどの精度に丸めるかを制御します。

値

p は次のいずれかでなければなりません。

表 A-12 -fprecision の値

p の値	意味
single	IEEE 単精度値に丸めます。
double	IEEE 倍精度値に丸めます。
extended	利用可能な最大の精度に丸めます。

p が single か double であれば、丸め精度モードは、プログラムの実行が始まるときに、それぞれ single か double 精度に設定されます。*p* が extended であるか、-fprecision フラグが使用されていなければ、丸め精度モードは extended 精度のままです。

single 精度の丸めモードでは、結果が 24 ビットの有効桁に丸められます。double 精度の丸めモードでは、結果が 53 ビットの有効桁に丸められます。デフォルトの extended 精度の丸めモードでは、結果が 64 ビットの有効桁に丸められます。この

モードは、レジスタにある結果をどの精度に丸めるかを制御するだけであり、レジスタの値には影響を与えません。レジスタにあるすべての結果は、拡張倍精度形式の全範囲を使って丸められます。ただし、メモリーに格納される結果は、指定した形式の範囲と精度に合わせて丸められます。

`float` 型の公称精度は `single` です。`long double` 型の公称精度は `extended` です。

デフォルト

`-fprecision` フラグを指定しないと、丸め精度モードは `extended` になります。

警告

このオプションは、IA プラットホームでメインプログラムをコンパイルするときしか有効ではありません。SPARC プラットホームでは、このオプションは無視されます。

-fround=*r*

起動時に IEEE 丸めモードを有効にします。

このオプションは、次に示す IEEE 754 丸めモードを設定します。

- 定数式を評価する時にコンパイラが使用できる。
- プログラム初期化中の実行時に設定される。

内容は、`ieee_flags` サブルーチンと同じです。これは実行時のモードを変更するために使用します。

値

r には次の値のいずれかを指定します。

表 A-13 `-fround` の値

r の値	意味
`nearest`	もっとも近い数値に丸め、中間値の場合は偶数にします。
`tozero`	ゼロに丸めます。
`negative`	負の無限大に丸めます。
`positive`	正の無限大に丸めます。

デフォルト

`-fround` オプションを指定しないと、丸めモードは `-fround=nearest` になります。

警告

1 つのルーチンを `-fround=`*r* でコンパイルした場合は、そのプログラムのすべてのルーチンを同じ `-fround=`*r* オプションでコンパイルする必要があります。そうしないと、予期しない結果が生じることがあります。

このオプションは、メインプログラムをコンパイルするときにだけ有効です。

`-fsimple[=`*n*`]`

浮動小数点最適化の設定を選択します。

このオプションで浮動小数点演算に影響する前提を設けることにより、オプティマイザで行う浮動小数点演算が簡略化されます。

値

n を指定する場合、0、1、2 のいずれかにしなければなりません。

表 A-14 `-fsimple` の値

n の値	意味
0	仮定の設定を許可しません。IEEE 754 に厳密に準拠します。
1	安全な簡略化を行います。その結果生成されたコードは、IEEE 754 に厳密には合致していませんが、大多数のプログラムの数値結果は変わりません。 `-fsimple=1` の場合、次に示す内容を前提とした最適化が行われます。 • IEEE 754 のデフォルトの丸めとトラップモードが、プロセスの初期化以後も変わらない。 • 起こり得る浮動小数点例外を除き、目に見えない結果を出す演算が削除される可能性がある。 • 無限大数または非数をオペランドとする演算は、その結果に非数を伝える必要がある。x*0 は 0 によって置き換えられる可能性がある。 • 演算はゼロの符号を区別しない。 `-fsimple=1` の場合、四捨五入や例外を考慮せずに完全な最適化を行うことは許可されていません。特に浮動小数点演算は、丸めモードを保持した定数について実行時に異なった結果を出す演算に置き換えることはできません。
2	これは浮動小数点演算の最適化を積極的に行い、丸めモードの変更によって多くのプログラムが異なった数値結果を出すようになります。たとえば、あるループ内の `x/y` の演算をすべて `x*z` に置き換えるような最適化を許可します。この最適化では、`x/y` はループ内で少なくとも 1 回評価されることが保証されており、`y` と `z` にはループの実行中に定数値が割り当てられます。

デフォルト

`-fsimple` を指定しないと、`-fsimple=0` が使用されます。

`-fsimple` を指定しても *n* の値を指定しないと、`-fsimple=1` が使用されます。

相互の関連性

`-fast` は `-fsimple=2` を意味します。

警告

このオプションによって、IEEE 754 に対する適合性が損なわれることがあります。

関連項目

`-fast`

-fstore

(IA) このオプションを指定すると、コンパイラは、次の場合に浮動小数点の式や関数の値を代入式の左辺の型に変換します。つまり、その値はレジスタにそのままの型で残りません。

- 式や関数を変数に代入する。
- 式をそれより短い浮動小数点型にキャストする。

このオプションを解除するには、オプション `-nofstore` を使用してください。

警告

丸めや切り捨てによって、結果がレジスタの値から生成される値と異なることがあります。

関連項目

`-nofstore`

-ftrap=*t*[,*t*...]

起動時に IEEE 754 トラップモードを有効に設定します。

このオプションは、プログラムの初期化時に設定される IEEE 754 トラップモードを設定しますが、`SIGFPE` ハンドラはインストールしません。トラップの設定と `SIGFPE` ハンドラのインストールを同時に行うには、`ieee_handler` を使用します。複数の値を指定すると、それらの値は左から右に処理されます。

値

t には次の値のいずれかを指定できます。

表 A-15 -ftrap の値

t の値	意味
[no%]division	ゼロによる除算をトラップします [しません]。
[no%]inexact	正確でない結果をトラップします [しません]。
[no%]invalid	無効な操作をトラップします [しません]。
[no%]overflow	オーバーフローをトラップします [しません]。
[no%]underflow	アンダーフローをトラップします [しません]。
%all	上のすべてをトラップします。
%none	上のどれもトラップしません。
common	無効、ゼロ除算、オーバーフローをトラップします。

[no%] 形式のオプションは、下の例に示すように、%all や common フラグの意味を変更するときだけ使用します。これは、特定のトラップを明示的に無効にするものではありません。

IEEE トラップを有効にする場合は、-ftrap=common の設定をお勧めします。

デフォルト

-ftrap を指定しないと、-ftrap=%none が使用されます (トラップは自動的には有効にされません)。

例

1 つ以上の値を指定すると、それらは左から右に処理されます。したがって、-ftrap=%all,no%inexact と指定すると、-inexact を除くすべてのトラップが設定されます。

相互の関連性

モードは、実行時に ieee_handler(3M) で変更できます。

警告

1 つのルーチンを -ftrap=*t* オプションでコンパイルした場合は、そのプログラムのルーチンすべてを -ftrap=*t* オプションを使用してコンパイルしてください。途中から異なるオプションを使用すると、予想に反した結果が生じることがあります。

-ftrap=inexact のトラップは慎重に使用してください。-ftrap=inexact では、浮動小数点の値が正確でないとトラップが発生します。たとえば、次の文ではこの条件が発生します。

```
x = 1.0 / 3.0;
```

このオプションは、メインプログラムをコンパイルするときにだけ有効です。このオプションを使用する際には注意してください。IEEE トラップを有効にするには -ftrap=common を使用してください。

関連項目

ieee_handler(3M) のマニュアルページ

-G

実行可能ファイルではなく動的共有ライブラリを構築します。

コマンド行で指定したソースファイルはすべて、デフォルトで -Kpic オプションでコンパイルされます。

テンプレートを使用する共有ライブラリを作成する場合は、通常、テンプレートデータベースでインスタンス化されているテンプレート関数を、共有ライブラリに組み込む必要があります。このオプションを使用すると、これらのテンプレートが必要に応じて共有ライブラリに自動的に追加されます。

相互の関連性

-c (コンパイルのみのオプション) を指定しないと、次のオプションが ld に渡されます。

- -dy
- -G
- -R

警告

共有ライブラリの構築には、ld -G ではなく、CC -G を使用してください。こうすると、CC ドライバによって C++ に必要ないくつかのオプションが ld に自動的に渡されます。

-G オプションを使用すると、コンパイラはデフォルトの -l オプションを ld に渡しません。共有ライブラリを別の共有ライブラリに依存させたい場合は、必要な -l オプションをコマンド行に渡す必要があります。たとえば、共有ライブラリを libCrun に依存させたい場合は、-lCrun をコマンド行に渡す必要があります。

関連項目

-dy、-Kpic、-xcode=pic13、-xildoff、-ztext、ld(1) のマニュアルページ、237 ページの「動的 (共有) ライブラリの構築」

-g

dbx(1) または Debugger によるデバッグおよびパフォーマンスアナライザ analyzer(1) による解析用のシンボルテーブル情報を追加生成します。

コンパイラとリンカーに、デバッグとパフォーマンス解析に備えてファイルとプログラムを用意するように指示します。

これには、次の処理が含まれています。

- オブジェクトファイルと実行可能ファイルのシンボルテーブル内に、詳細情報 (スタブ) を生成する。
- 「支援関数」を生成する。デバッガはこれを呼び出して、デバッガの機能のいくつかを実現する。
- 関数のインライン生成を無効にする。
- 特定のレベルの最適化を無効にする。

相互の関連性

このオプションと -xo*level* （あるいは、同等の -O オプションなど）を一緒に使用した場合、デバッグ情報が限定されます。詳細は、375 ページの「-xolevel」を参照してください。

このオプションを使用するとき、最適化レベルが -xO3 以下の場合、可能な限りのシンボリック情報とほぼ最高の最適化が得られます。末尾呼び出しの最適化とバックエンドのインライン化は無効です。

このオプションを使用するとき、最適化レベルが -xO4 以上の場合、可能な限りのシンボリック情報と最高の最適化が得られます。

このオプションを指定すると、+d オプションが自動的に指定されます。

このオプションを指定すると、-xildon が指定されてデフォルトのリンカーがインクリメンタルリンカーのオプションになるため、コンパイル、編集、デバッグのサイクルを効率的に実行できます。

次の条件のどれかが真でない場合は、ld ではなく ild が起動されます。

- -G オプションを指定している
- -xildoff オプションを指定している
- コマンド行でソースファイルを指定している

パフォーマンスアナライザの機能を最大限に利用するには、-g オプションを指定してコンパイルします。一部のパフォーマンス解析機能では、-g オプションを必要としませんが、注釈付きのソース、一部の関数レベル情報、およびコンパイラの注釈メッセージを表示するには -g を指定してコンパイルする必要があります。詳細は、analyzer(1) のマニュアルページと『プログラムのパフォーマンス解析』を参照してください。

-g を指定して生成された注釈メッセージでは、プログラムのコンパイル中にコンパイラで行われた最適化や変換について説明します。メッセージを表示するには、er_src(1) コマンドを使用します。これらのメッセージはソースコードでインタリーブされます。

警告

プログラムを別々の手順でコンパイルしてリンクしてから 1 つの手順に -g オプションを取り込み、他の手順から -g オプションを除外すると、プログラムの正確さは損なわれませんが、プログラムをデバッグする機能には影響を与えます。-g (または -g0) でコンパイルされていなくて、-g (または -g0) とリンクされているモジュールは、デバッグ用に正しく作成されません。-g オプション (または -g0 オプション) を指定した main 関数の入っているモジュールのコンパイルは通常デバッグに必要です。

関連項目

+d、-g0、-xildoff、-xildon、-xs、および analyzer(1) 、er_src(1)、ld(1) のマニュアルページ、『dbx コマンドによるデバッグ』(スタブの詳細について)、『プログラムのパフォーマンス解析』

-g0

デバッグ用のコンパイルとリンクを行いますが、インライン展開は行いません。

このオプションは、+d が有効化されないという点を除いて、-g と同じです。

関連項目

+d、-g、-xildon、『dbx コマンドによるデバッグ』

-H

インクルードしたファイルのパス名を印します。

現在のコンパイルに含まれている #include ファイルのパス名を標準エラー出力 (stderr) に 1 行に 1 つずつ出力します。

-h[ ]*name*

生成する動的共有ライブラリに名前 *name* を割り当てます。

これはローダー用のオプションで、ld に渡されます。通常、-h の後に指定する *name* (名前) は、-o の後に指定する名前と同じでなければなりません。-h と *name* の間には、空白文字を入れても入れなくてもかまいません。

コンパイルの時ローダーは、作成対象の共有動的ライブラリに、指定の名前を割り当てます。この名前は、ライブラリのイントリンシック名として、ライブラリファイルに記録されます。-h *name* オプションがない場合、イントリンシック名はライブラリファイルに記録されません。

実行可能ファイルはすべて、必要な共有ライブラリファイルのリストを持っています。実行時のリンカーは、ライブラリを実行可能ファイルにリンクするとき、ライブラリのイントリンシック名をこの共有ライブラリファイルのリストの中にコピーします。共有ライブラリにイントリンシック名がないと、リンカーは代わりにその共有ライブラリファイルのパス名を使用します。

例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

-help

-xhelp=flags と同じです。

-I*pathname*

#include ファイル検索パスに *pathname* を追加します。

このオプションは、インクルードファイルの相対ファイル名 (スラッシュ以外の文字で始まるファイル名) リストに、*pathname* (パス名) を追加します。

コンパイラでは、引用符をインクルードした (#include "foo.h" 形式の) ファイルを次の順序で検索します。

1. ソースが存在するディレクトリ
2. -I オプションで指定したディレクトリ内 (存在する場合)
3. コンパイラで提供される C++ ANSI C ヘッダファイル、および特殊な目的なファイル内ヘッダファイルの include ディレクトリ
4. /usr/include ディレクトリ内

コンパイラでは、山括弧をインクルードした (#include <foo.h> 形式の) ファイルを次の順序で検索します。

1. -I オプションで指定したディレクトリ内 (存在する場合)
2. コンパイラで提供される C++ ANSI C ヘッダファイル、および特殊な目的なファイル内ヘッダファイルの include ディレクトリ

3. /usr/include ディレクトリ内

注 – スペルが標準ヘッダーファイルの名前と一致する場合は、171 ページの「標準ヘッダーの実装」 も参照してください。

相互の関連性

-I- オプションを指定すると、デフォルトの検索規則が無効になります。

-library=no%Cstd を指定すると、その検索パスに C++ 標準ライブラリに関連付けられたコンパイラで提供されるヘッダーファイルがコンパイラでインクルードされません。169 ページの「C++ 標準ライブラリの置き換え」を参照してください。

-pti*path* が使用されていないと、コンパイラは -I*pathname* でテンプレートファイルを探します。

-pti*path* の代わりに -I*pathname* を使用します。

このオプションは、置き換えられる代わりに蓄積されます。

関連項目

-I-

-I-

インクルードファイルの検索規則を次のとおり変更します。

#include "foo.h" 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I オプションで指定されたディレクトリ内 (-I- の前後)
2. コンパイラで提供される C++ ヘッダーファイルの include ディレクトリ、ANSI C ヘッダーファイル、および特殊な目的なファイル内
3. /usr/include ディレクトリ内

#include <foo.h> 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I- の後に指定した -I オプションで指定したディレクトリ内
2. コンパイラで提供される C++ ヘッダーファイルの include ディレクトリ、ANSI C ヘッダーファイル、および特殊な目的なファイル内
3. /usr/include ディレクトリ内

注 – インクルードファイルの名前が標準ヘッダーファイルの名前と一致する場合は、171 ページの「標準ヘッダーの実装」 も参照してください。

例

次の例は、prog.cc のコンパイル時に-I- を使用した結果を示します。

```
prog.cc       #include "a.h"
              #include <b.h>
              #include "c.h"

c.h           #ifndef _C_H_1
              #define _C_H_1
              int c1;
              #endif

inc/a.h       #ifndef _A_H
              #define _A_H
              #include "c.h"
              int a;
              #endif

inc/b.h       #ifndef _B_H
              #define _B_H
              #include <c.h>
              int b;
              #endif

inc/c.h       #ifndef _C_H_2
              #define _C_H_2
              int c2;
              #endif
```

次のコマンドでは、#include "foo.h" 形式のインクルード文のカレントディレクトリ (インクルードしているファイルのディレクトリ) のデフォルトの検索動作を示します。#include "c.h" ステートメントを inc/a.h で処理するときは、コンパイラでinc サブディレクトリからc.h ヘッダーファイルがインクルードされます。#include "c.h" 文を prog.cc で処理するときは、コンパイラで prog.cc の入っているディレクトリから c.h ファイルがインクルードされます。-H オプションがインクルードファイルのパスを印刷するようにコンパイラに指示していることに注意してください。

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

次のコマンドでは、-I- オプションの影響を示します。コンパイラでは、#include "foo.h" 形式の文を処理するときにインクルードしているディレクトリを最初に探しません。その代わり、-I オプションで名前の付いたディレクトリをコマンド行に表示された順序で検索します。inc/a.h の #include "c.h" 文を処理するときは、コンパイラには inc/c.h ヘッダファイルの代わりに ./c.h ヘッダファイルがインクルードされます。

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

相互の関連性

-I- がコマンド行に表示されると、コンパイラではディレクトリが -I 指示に明示的に表示されていない限り決してカレントディレクトリを検索しません。この影響は #include "foo.h" 形式のインクルード文にも及びます。

警告

コマンド行の最初の -I- だけが、説明した動作を引き起こします。

-i

リンカー ld は LD_LIBRARY_PATH の設定を無視します。

-inline

-xinline と同じです。

-instances=*a*

テンプレートインスタンスの位置とリンケージを制御します。

値

a には次のいずれかを指定します。

表 A-16 -instances の値

a の値	意味
explicit	明示的にインスタンス化されたインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。必要なインスタンスがほかにあっても生成しません。
extern	必要なすべてのインスタンスをテンプレートリポジトリに置き、それらに対して大域リンケージを行います (リポジトリのインスタンスが古い場合は、再びインスタンス化されます)。
global	必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。
semiexplicit	明示的にインスタンス化されたインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。明示的なインスタンスにとって必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。必要なインスタンスがほかにあっても生成しません。
static	必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して静的リンケージを行います。

デフォルト

`-instances` を指定しないと、`-instances=global` が使用されます。

関連項目

87 ページの「テンプレートインスタンスの配置とリンケージ」

-instlib=<出力ファイル>

このオプションを使用すると、ライブラリ (共有、静的) と現在のオブジェクトで重複するテンプレートインスタンスの生成が禁止されます。一般に、ライブラリを使用するプログラムが多数のインスタンスを共有する場合、`-instlib=`*filename* を指定して、コンパイル時間の短縮を試みることができます。

値

既存のテンプレートインスタンスが入っていることがわかっているライブラリを指定するには、*filename* 引数を使用します。ファイル名引数には、スラッシュ (/) 文字を含める必要があります。現在のディレクトリを基準とする相対パスの場合には、ドット・スラッシュ (./) を使用します。

デフォルト

`-instlib=`*filename* オプションにはデフォルト値はないので、値を指定する場合だけに使用します。このオプションは複数回指定することができ、指定内容は追加されていきます。

例:

`libfoo.a` ライブラリと `libbar.so` ライブラリが、ソースファイル `a.cc` と共有する多数のテンプレートインスタンスをインスタンス化すると仮定します。
`-instlib=`*filename* を追加してライブラリを指定すると、冗長性が回避されコンパイル時間を短縮できます。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

相互の関連性

-g を使ってコンパイルするとき、-instlib=*filename* で指定したライブラリが -g でコンパイルされていない場合には、テンプレートインスタンスがデバッグ不能となります。この問題の対策としては、-g を指定するときに -instlib=*filename* を使用しないようにします。

-L パスにおいて指定ライブラリが検索されることはありません。

警告:

-instlib によってライブラリを指定する場合には、そのライブラリとのリンクを行う必要があります。

関連項目

-template, -instances, -pti

-KPIC

SPARC: -xcode=pic32. と同じです。

IA:-Kpic と同じです。

このオプションは、共有ライブラリを構築するためにソースファイルをコンパイルするときに使用します。大域データへの参照は、それぞれ大域オフセットテーブルでのポインタの間接参照として生成されます。各関数呼び出しは、手続きリンケージテーブルを通して PC 相対アドレス指定モードで生成されます。

-Kpic

SPARC: -xcode=pic13. と同じです。

IA:位置に依存しないコードを使ってコンパイルします。

このオプションは、共有ライブラリを構築するためにソースファイルをコンパイルするときに使用します。大域データへの参照は、それぞれ大域オフセットテーブルでのポインタの間接参照として生成されます。各関数呼び出しは、手続きリンケージテーブルを通して PC 相対アドレス指定モードで生成されます。

-keeptmp

コンパイル中に作成されたすべての一時ファイルを残しておきます。

このオプションを -verbose=diags と一緒に使用すると、デバッグに便利です。

関連項目

-v, -verbose

-L*path*

ライブラリを検索するディレクトリに、*path* ディレクトリを追加します。

このオプションは ld に渡されます。コンパイラが提供するディレクトリよりも*path* が先に検索されます。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

-l*lib*

ライブラリ lib*lib*.a または lib*lib*.so をリンカーの検索ライブラリに追加します。

このオプションは ld に渡されます。通常のライブラリは、名前が lib*lib*.a か lib*lib*.so の形式です (lib と .a または .so の部分は必須です)。このオプションでは*lib* の部分を指定できます。コマンド行には、ライブラリをいくつでも指定できます。指定したライブラリは、-L*dir* で指定された順に検索されます。

-l*lib* オプションはファイル名の後に指定してください。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

正しい順序でライブラリが検索されるようにするには、安全のため、必ずソースとオブジェクトの後に -l*x* を使用してください。

警告

libthread とリンクする場合は、ライブラリを正しい順序でリンクするために -lthread ではなく -mt を使用してください。

POSIX スレッドを使用する場合は、-mt オプションと -lpthread オプションを使ってリンクする必要があります。-mt オプションが必要な理由は、libCrun (標準モード) と libC (互換モード) がマルチスレッド対応のアプリケーションに対して libthread を必要とするためです。

関連項目

-L*dir*、-mt、第 12 章、『Tools.h++ クラスライブラリ・リファレンスマニュアル』

-libmieee

-xlibmieee と同じです。

-libmil

-xlibmil と同じです。

-library=*l*[,*l*...]

l に指定した、CC が提供するライブラリを、コンパイルとリンクに組み込みます。

値

互換モード（-compat[=4]）の場合、*l* には次のいずれかを指定します。

表 A-17 互換モードに使用できる-library オプション

l の値	意味
[no%]f77	非推奨。使用しないでください。-xlang=f77 を使用してください。
[no%]f90	非推奨。使用しないでください。-xlang=f90 を使用してください。
[no%]f95	非推奨。使用しないでください。-xlang=f95 を使用してください。
[no%]rwtools7	古い Tools.h++ バージョン 7 を使用します [しません]。
[no%]rwtools7_dbg	デバッグ可能な Tools.h++ バージョン 7を使用します [しません]。
[no%]complex	複素数の演算に libcomplex を使用します [しません]。
[no%]interval	非推奨。使用しないでください。-xia を使用してください。
[no%]libC	C++ サポートライブラリ libC を使用します [しません]。
[no%]gc	ガベージコレクション libgc を使用します [しません]。
[no%]sunperf	*SPARC*:Sun Performance Library™ を使用します [しません]。
%all	非推奨。-library=%all は -library=f77、f90、rwtools7、complex、interval、gc を指定する場合と同じです。libC ライブラリは、-library=no%libC により特に除外されていない限り必ず取り込んでください。詳細は、「警告」の項を参照してください。
%none	libC の場合を除いてC++ ライブラリを一切使用しません。

標準モード (デフォルトモード) の場合、*l* には次のいずれかを指定します。

表 A-18 標準モードに使用できる `-library` オプション

l の値	意味
`[no%]f77`	非推奨。使用しないでください。`-xlang=f77` を使用してください。
`[no%]f90`	非推奨。使用しないでください。`-xlang=f90` を使用してください。
`[no%]f95`	非推奨。使用しないでください。`-xlang=f95` を使用してください。
`[no%]rwtools7`	古い `Tools.h++` バージョン 7 を使用します [しません]。
`[no%]rwtools7_dbg`	デバッグ可能な `Tools.h++` バージョン 7を使用します [しません]。
`[no%]rwtools7_std`	標準 iostream `Tools.h++` バージョン 7 を使用します [しません]。
`[no%]rwtools7_std_dbg`	デバッグが可能な標準 `iostream Tools.h++` バージョン 7 を使用します [しません]。
`[no%]interval`	非推奨。使用しないでください。`-xia` を使用してください。
`[no%]iostream`	古い `iostream` ライブラリ `libiostream` を使用します [しません]。
`[no%]Cstd`	C++ 標準ライブラリ `libCstd` を使用します [しません]。コンパイラ付属の C++ 標準ライブラリヘッダーファイルをインクルードします [しません]。
`[no%]Crun`	C++ 実行時ライブラリ `libCrun` を使用します [しません]。
`[no%]gc`	ガベージコレクション `libgc` を使用します [しません]。
`[no%]stlport4`	デフォルトの `libCstd` の代わりに STLport の標準ライブラリのバージョン 4.5.3 を使用します [しません]。STLport の詳しい使用方法については、193 ページの「STLport」を参照してください。
`[no%]stlport4_dbg`	STLport のデバッグ可能なライブラリを使用します [しません]。

表 A-18 標準モードに使用できる -library オプション (続き)

/ の値	意味
[no%]sunperf	*SPARC*:Sun Performance Library™ を使用します [しません]。
%all	非推奨。-library=%all は -library=f77、f90、rwtools7、gc、interval、iostream、Cstd を指定する場合と同じです。libCrun ライブラリは、-library=no%Crun により特に除外されていない限り必ず取り込んでください。詳細は、「警告」の項を参照してください。
%none	libCrun の場合を除いてC++ ライブラリを使用しません。

デフォルト

- **互換モード (**-compat[=4]**)**
  - -library を指定しない場合は、-library=%none が使用されます。
  - -library=%none または-library=no%libC で特に除外されない限り、libC ライブラリは常に含まれます。
- **標準モード (デフォルトモード)**
  - -library を指定しない場合は、-library=%none,Cstd が使用されます。
  - -library=%none、-library=no%Cstd、-library=stlport4 のいずれかで特に除外されない限り、libCstd ライブラリは常に含まれます。
  - -library=no%Crun で特に除外されない限り、libCrun ライブラリは常に含まれます。

例

標準モードで libCrun 以外の C++ ライブラリを除外してリンクするには、次のコマンドを使用します。

```
example% CC -library=%none
```

標準モードで従来の iosttream と RogueWave tools.h++ ライブラリを使用するには、次のコマンドを使用します。

```
example% CC -library=rwtools7,iostream
```

標準モードで標準 の iostream と Rogue Wave tools.h++ ライブラリを使用するコマンドは次のとおりです。

```
example% CC -library=rwtools7_std
```

互換モードで従来の iostream と Rogue Wave tools.h++ ライブラリを使用するコマンドは次のとおりです。

```
example% CC -compat -library=rwtools7
```

相互の関連性

-library でライブラリを指定すると、適切な -I パスがコンパイルで設定されます。リンクでは、適切な -L、-Y P および、-R パスと、-l オプションが設定されます。

このオプションは、置き換えられる代わりに蓄積されます。

区間演算ライブラリを使用するときは、libC、libCstd、または libiostream のいずれかのライブラリを取り込む必要があります。

-library オプションを使用すると、指定したライブラリに対する -l オプションが正しい順序で送信されるようになります。たとえば、-library=rwtools7,iostream および -lirabary=iostream,rwtools7 のどちらでも、-l オプションは、-lrwtool -liostream の順序で ld に渡されます。

指定したライブラリは、システムサポートライブラリよりも前にリンクされます。

-library=sunperf と -xlic_lib=sunperf は同じコマンド行で使用できません。

-library=stlport4 と -library=Cstd を同一のコマンド行で使用できません。

同時に使用できる RogueWave ツールライブラリは 1 つだけです。また、`-library=stlport4` を指定して RogueWave ツールライブラリと併用することはできません。

従来の iostream RogueWave ツールライブラリを標準モード (デフォルトモード) で取り込む場合は、`libiostream` も取り込む必要があります (詳細は、『C++ 移行ガイド』を参照してください)。標準 iostream RogueWave ツールライブラリは、標準モードでのみ使用できます。次のコマンド例は、RogueWave `tools.h++` ライブラリオプションの有効もしくは無効な使用法について示します。

```
% CC -compat -library=rwtools foo.cc          <-- valid
% CC -compat -library=rwtools_std foo.cc      <-- invalid

% CC -library=rwtools,iostream foo.cc         <-- valid, classic iostreams
% CC -library=rwtools foo.cc                  <-- invalid

% CC -library=rwtools_std foo.cc              <-- valid, standard iostreams
% CC -library=rwtools_std,iostream foo.cc     <-- invalid
```

`libCstd` と `libiostream` の両方を含めた場合は、プログラム内で新旧両方の形式の `iostream` (例: `cout` と `std::cout`) を使用して、同じファイルにアクセスしないよう注意してください。同じプログラム内に標準 `iostream` と従来の `iostream` が混在し、その両方のコードから同じファイルにアクセスすると、問題が発生する可能性があります。

`libC` とも `libCrun` ともリンクしないプログラムは、C++ のすべての機能を使用できないことがあります。

`-xnolib` を指定すると、`-library` は無視されます。

警告

別々の手順でコンパイルしてリンクする場合は、コンパイルコマンドに表示される一連の `-library` オプションをリンクコマンドにも表示する必要があります。

`stlport4`、`Cstd`、および `iostream` のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を `-library` オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。STLport の詳しい使用方法については、193 ページの「STLport」を参照してください。

これらのライブラリは安定したものではなく、リリースによって変わることがあります。

`-library=%all` オプションの使用は次の理由からお勧めしません。

- このコマンドを使用して取り込まれる正確な一連のライブラリがリリースごとに変わる可能性がある。
- 予期していたライブラリを取得できない可能性がある。
- 予期していないライブラリを取得する可能性がある。
- makefile コマンド行を見る他の開発者が、どれをリンクしようとしていたかがわからない。
- このオプションはコンパイラの将来のリリースで削除される。

関連項目

`-I`、`-l`、`-R`、`-staticlib`、`-xia`、`-xlang`、`-xnolib`、第 12 章、第 13 章、第 14 章、32 ページの「標準ライブラリヘッダーファイルに対する `make` の使用」、『Tools.h++ ユーザーズガイド』、『Tools.h++ クラスライブラリ・リファレンスマニュアル』、『Standard C++ Library Class Reference』(英語版のみ)、『C++ Interval Arithmetic Programming Reference』(英語版のみ)

`-library=no%cstd` オプションを使用して、ユーザー独自の C++ 標準ライブラリの使用を有効にする方法については、169 ページの「C++ 標準ライブラリの置き換え」を参照してください。

-mc

オブジェクトファイルの .comment セクションから重複文字列を削除します。文字列に空白が含まれている場合は、文字列を引用符で囲む必要があります。`-mc` オプションを使用すると、`mcs -c` コマンドが呼び出されます。

-migration

以前のバージョンのコンパイラ用に作成されたソースコードの移行に関する情報の参照先を表示します。

注 – このオプションは次のリリースでは存在しなくなる可能性があります。

-misalign

SPARC:通常はエラーになる、メモリー中の境界整列の誤ったデータを許可します。以下に例を示します。

```
char b[100];
int f(int * ar) {
return *(int *) (b +2) + *ar;
}
```

このオプションは、プログラムの中に正しく境界整列されていないデータがあることをコンパイラに知らせます。したがって、境界整列が正しくない可能性があるデータに対しては、ロードやストアを非常に慎重に (つまり、1 度に 1 バイトずつ) 行う必要があります。このオプションを使用すると、実行速度が大幅に低下することがあります。低下する程度はアプリケーションによって異なります。

相互の関連性

SPARC プラットフォームで `#pragma pack` を使って、型のデフォルト境界整列よりも高い密度でデータをパックする場合は、アプリケーションのコンパイルとリンクに `-misalign` オプションを指定する必要があります。

境界整列が正しくないデータは、実行時に `ld` のトラップ機構によって処理されます。`misalign` オプションとともに最適化フラグ (`-x0{1|2|3|4|5}` またはそれと同等のフラグ) を使用すると、ファイル境界整列の正しくないデータを正しい境界に整列に合わせるための命令がオブジェクトに挿入されます。この場合には、実行時不正境界整列トラップは生成されません。

警告

できれば、プログラムの境界整列が正しい部分と境界整列が誤った部分をリンクしないでください。

コンパイルとリンクを別々に行う場合は、`-misalign` オプションをコンパイルコマンドとリンクコマンドの両方で指定する必要があります。

-mr[,*string*]

オブジェクトファイルの .comment セクションからすべての文字列を削除します。*string* が与えられた場合、そのセクションに *string* を埋め込みます。文字列に空白が含まれている場合は、文字列を引用符で囲む必要があります。このオプションを使用すると、mcs -d [-a *string*] が呼び出されます。

相互の関連性

このオプションは、-S、-xsbfast、または -sbfast が指定されると無効になります。

-mt

マルチスレッド化したコードのコンパイルとリンクを行います。

このオプションでは、次のことが行われます。

- -D_REENTRANT をプリプロセッサに渡します。
- -lthread を正しい順序で ld に渡します。
- 標準モード (デフォルトモード) では、libthread が libCrun よりも前にリンクされるようにします。
- 互換モード (-compat) では、libthread が libC よりも前にリンクされるようにします。

アプリケーションやライブラリがマルチスレッド化されている場合は、-mt オプションが必要です。

警告

libthread とリンクする場合には、-lthread ではなく -mt オプションを使用してライブラリのリンク順序が正しくなるようにしてください。

POSIX スレッドを使用する場合は、-mt オプションと -lpthread オプションを使ってリンクする必要があります。-mt オプションが必要な理由は、libCrun (標準モード) と libC (互換モード) がマルチスレッド対応のアプリケーションに対して libthread を必要とするためです。

コンパイルとリンクを別々に実行する場合で、コンパイルで -mt を使用した場合は、次の例に示すようにリンクでも -mt を使用してください。そうしないと、予期しない結果が発生する可能性があります。

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

並列の Fortran オブジェクトを C++ オブジェクトと混合している場合は、リンク行に -mt オプションを指定する必要があります。

関連項目

-xnolib、第 11 章、『マルチスレッドのプログラミング』、『リンカーとライブラリ』

-native

-xtarget=native と同じです。

-noex

-features=no%except と同じです。

-nofstore

IA:強制された式の精度を無効にします。

このオプションを指定すると、次のどちらの場合でも、コンパイラは浮動小数点の式や関数の値を代入式の左辺の型に変換しません。つまり、レジスタの値はそのままです。

- 式や関数を変数に代入する

標準モード (デフォルトモード) では、libthread が libCrun よりも前にリンクされるようにします。

- 式や関数をそれより短い浮動小数点型にキャストする

関連項目

-fstore

-nolib

-xnolib と同じです。

-nolibmil

-xnolibmil と同じです。

-noqueue

ライセンスを待ち行列に入れません。

ライセンスを確保できない場合、コンパイラはコンパイル要求を待ち行列に入れず、コンパイルもしないで終了します。makefile のテストには、ゼロ以外の状態が返されます。

-norunpath

実行可能ファイルに共有ライブラリへの実行時検索パスを組み込みません。

実行可能ファイルが共有ライブラリを使用する場合、コンパイラは通常、実行時のリンカーに対して共有ライブラリの場所を伝えるために構築を行なったパス名を知らせます。これは、-ld に対して -R オプションを渡すことによって行われます。このパスはコンパイラのインストール先によって決まります。

このオプションは、プログラムで使用される共有ライブラリへのパスが異なる顧客に出荷される実行可能ファイルの構築にお勧めします。

相互の関連性

共有ライブラリをコンパイラのインストールされている位置 (デフォルトのインストール先は /opt/SUNWspro/lib) で使用し、かつ -norunpath を使用する場合は、リンク時に -R オプションを使うか、または実行時に環境変数

LD_LIBRARY_PATH を設定して共有ライブラリの位置を明示しなければなりません。そうすることにより、実行時リンカーはその共有ライブラリを見つけることができます。

-O

-xO2 と同じです。

-O*level*

-x*Olevel.* と同じです。

-o *filename*

出力ファイルまたは実行可能ファイルの名前を *filename* (ファイル名) に指定します。

相互の関連性

コンパイラは、テンプレートインスタンスを格納する必要がある場合には、出力ファイルのディレクトリにあるテンプレートリポジトリに格納します。たとえば、次のコマンドでは、コンパイラはオブジェクトファイルを ./sub/a.o に、テンプレートインスタンスを ./sub/SunWS_cache 内のリポジトリにそれぞれ書き込みます。

```
example% CC -o sub/a.o a.cc
```

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。たとえば、次のコマンドでは、コンパイラは ./sub1/SunWS_Cache と ./sub2/SunWS_cache から読み取り、必要な場合は ./SunWS_cache に書き込みます。

```
example% CC sub1/a.o sub2/b.o
```

詳細は、92 ページの「テンプレートリポジトリ」を参照してください。

警告

このファイル名には、コンパイラが作成するファイルの型に合った接尾辞を指定してください。また、cc ドライバはソースファイルには上書きしないため、ソースファイルとは異なるファイルを指定する必要があります。

+p

標準に従っていないプリプロセッサの表明を無視します。

デフォルト

+p を指定しないと、コンパイラは非標準のプリプロセッサの表明を認識します。

相互の関連性

+p を指定している場合は、次のマクロは定義されません。

- sun
- unix
- sparc
- i386

-P

ソースの前処理だけでコンパイルはしません (接尾辞.iのファイルを出力します)。

このオプションを指定すると、プリプロセッサが出力するような行番号情報はファイルに出力されません。

関連項目

-E

-p

prof でプロファイル処理するためのデータを収集するオブジェクトコードを作成します。

-p は実行内容を記録し、正常終了時に mon.out というファイルを生成します。

警告

別々の手順でコンパイルしてリンクする場合は、-p オプションをコンパイルコマンドとリンクコマンドの両方に表示する必要があります。1 つの手順で -p を取り込み、もう 1 つの手順で除外すると、プログラムの正確さは損なわれませんがプロファイルを行えなくなります。

関連項目

-xpg、-xprofile、analyzer(1) のマニュアルページ、『プログラムのパフォーマンス解析』

-pentium

IA:-xtarget=pentium と置き換えられています。

-pg

-xpg と同じです。

-PIC

SPARC: -xcode=pic32. と同じです。

IA:-Kpic と同じです。

-pic

SPARC:-xcode=pic13 と同じです。

IA:-Kpic と同じです。

-pta

-template=wholeclass と同じです。

-pti*path*

テンプレートソース用の検索ディレクトリを追加指定します。

このオプションは -I*pathname* (パス名) によって設定された通常の検索パスに代わるものです。-pti*path* (パス) フラグを使用した場合、コンパイラはこのパス上にあるテンプレート定義ファイルを検索し、-I*pathname* フラグを無視します。

-pti*path* よりも -I*pathname* を使用すると混乱が起きにくくなります。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

関連項目

-I*pathname*

-pto

-instances=static と同じです。

-ptr

このオプションは廃止されたため、コンパイル時には無視されます。

警告

-ptr オプションは存在しても無視されますが、すべてのコンパイルコマンドから削除するようにしてください。これは将来のリリースで、-ptr が以前とは異なる動作のオプションとして再実装される可能性があるためです。

関連項目

リポジトリのディレクトリについては、92 ページの「テンプレートリポジトリ」を参照してください。

-ptv

-verbose=template と同じです。

-Qoption *phase option*[,*option*...]

option (オプション) を *phase* (コンパイル段階) に渡します。

複数のオプションを渡すには、コンマで区切って指定します。

値

phase には、以下の値のいずれか 1 つを指定します。

表 A-19 -Qoption の値

SPARC	IA
ccfe	ccfe
iropt	cg386
cg	codegen
CClink	CClink
ld	ld

例

次に示すコマンド行では、ld が CC ドライバによって起動されたとき、-Qoption で指定されたオプションの -i と -m が ld に渡されます。

```
example% CC -Qoption ld -i,-m test.c
```

警告

意図しない結果にならないように注意してください。そのためには、次のような方法があります。

```
-Qoption ccfe -features=bool,iddollar
```

しかしこの指定は、意図に反して次のように解釈されてしまいます。

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

正しい指定は次のとおりです。

```
-Qoption ccfe -features=bool,-features=iddollar
```

-qoption *phase option*

-Qoption と同じです。

-qp

-p と同じです。

-Qproduce *sourcetype*

CC ドライバに *sourcetype* (ソースタイプ) 型のソースコードを生成するよう指示します。

sourcetype に指定する接尾辞の定義は次のとおりです。

表 A-20 -Qproduce の値

接尾辞	意味
.i	ccfe が作成する前処理済みの C++ のソースコード
.o	コードジェネレータが作成するオブジェクトファイル
.s	cg が作成するアセンブラソース

-qproduce *sourcetype*

-Qproduce と同じです。

-R*pathname*[*:pathname...*]

動的ライブラリの検索パスを実行可能ファイルに組み込みます。

このオプションは ld に渡されます。

デフォルト

-R オプションを指定しないと、出力オブジェクトに記録され、実行時リンカーに渡されるライブラリ検索パスは、-xarch オプションで指定されたターゲットアーキテクチャ命令によって異なります (-xarc を指定しないと、-xarch=generic が使用されます)。

-xarch の値	デフォルトのライブラリ検索パス
v9、v9a、v9b	*install-directory*/SUNWspro/lib/v9
All other values	*install-directory*/SUNWspro/lib

標準インストールでは、*install-directory* は /opt です。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

LD_RUN_PATH 環境変数が設定されている場合に、-R オプションを指定すると、-R に指定したパスが検索され、LD_RUN_PATH のパスは無視されます。

関連項目

-norunpath、『リンカーとライブラリ』

-readme

-xhelp=readme と同じです。

-S

コンパイルしてアセンブリコードだけを生成します。

CC ドライバはプログラムをコンパイルして、アセンブリソースファイルを作成します。しかし、プログラムのアセンブルは行いません。このアセンブリソースファイル名には、.s という接尾辞が付きます。

-s

実行可能ファイルからシンボルテーブルを取り除きます。

出力する実行可能ファイルからシンボリック情報をすべて削除します。このオプションは ld に渡されます。

-sb

-xsb で置き換えられています。

-sbfast

-xsbfast と同じです。

-staticlib=*l*[,*l*...]

-library オプションで指定されている C++ ライブラリ (そのデフォルトも含む)、-xlang オプションで指定されているライブラリ、-xia オプションで指定されているライブラリのうち、どのライブラリが静的にリンクされるかを指定します。

値

l には、以下の値のいずれか 1 つを指定します。

表 A-21 `-staticlib` の値

l の値	意味
`[no%]` *library*	*library* を静的にリンクします [しません]。*library* に有効な値は、`-library` で有効なすべての値 (`%all` と `%none` を除く)、`-xlang` で有効なすべての値、および (`-xia` に関連して使用される) `interval` です。
`%all`	`-library` オプションで指定されているすべてのライブラリと、`-xlang` オプションで指定されているすべてのライブラリ、`-xia` をコマンド行で指定している場合は区間ライブラリを静的にリンクします。
`%none`	`-library` オプションと `-xlang` オプションに静的に指定されているライブラリをリンクしません。`-xia` をコマンド行に指定した場合は、区間ライブラリを静的にリンクしません。

デフォルト

`-staticlib` を指定しないと、`-staticlib=%none` が使用されます。

例

`-library` のデフォルト値は `Crun` であるため、次のコマンド行は、`libCrun` を静的にリンクします。

```
example% CC -staticlib=Crun (正しい)
```

これに対し、次のコマンド行は `libgc` をリンクしません。これは、`-library` オプションで明示的に指定しない限り、`libgc` はリンクされないためです。

```
example% CC -staticlib=gc (誤り)
```

libgc を静的にリンクするには、次のコマンドを使用します。

```
example% CC -library=gc -staticlib=gc (正しい)
```

次のコマンドは、librwtool ライブラリを動的にリンクします。librwtool はデフォルトのライブラリでもなく、-library オプションでも選択されていないため、-staticlib の影響はありません。

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 (誤り)
```

次のコマンドは、librwtool ライブラリを静的にリンクします。

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (正しい)
```

次のコマンドは、Sun Performance Library を動的にリンクします。これは、-staticlib オプションを Sun Performance Library のライブラリのリンクに反映させるために -library=sunperf を -staticlib=sunperf に関連させて使用する必要があるからです。

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (誤り)
```

次のコマンドは、Sun Performance Library を静的にリンクします。

```
example% CC -library=sunperf -staticlib=sunperf (正しい)
```

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

-staticlib オプションは、-xia、-xlang および -library オプションで明示的に選択された C++ ライブラリ、または、デフォルトで暗黙的に選択された C++ ライブラリだけに機能します。互換モードでは (-compat=[4])、libC がデフォルトで選択されます。標準モードでは (デフォルトのモード)、Cstd と Crun がデフォルトで選択されます。

-xarch=v9、-xarch=v9a、-xarch=v9bのいずれかを（あるいは、64 ビットアーキテクチャのオプションと同等のオプション）使用する場合、静的ライブラリとしては使用できない C++ ライブラリがあります。

警告

ライブラリで使用できる値は安定したものではないため、リリースによって変わることがあります。

関連項目

-library、166 ページの「標準ライブラリの静的リンク」

-temp=*path*

一時ファイルのディレクトリを定義します。

コンパイル中に生成される一時ファイルを格納するディレクトリのパス名を指定します。

関連項目

-keeptmp

-template=*opt*[,*opt*...]

さまざまなテンプレートオプションを有効/無効にします。

値

opt は次のいずれかの値である必要があります。

表 A-22 `-template` の値

c の値	意味
`[no%]extdef`	別のソースファイルからテンプレート定義を検索します [しません]。
`[no%]geninlinefuncs`	明示的にインスタンス化されたクラステンプレートのための非参照インラインメンバー関数を生成します [しません]。
`[no%]wholeclass`	コンパイラに対し、使用されている関数だけインスタンス化するのではなく、テンプレートクラス全体をインスタンス化する [しない] ように指示します。クラスの少なくとも 1 つのメンバーを参照しなければなりません。そうでない場合は、コンパイラはそのクラスのどのメンバーもインスタンス化しません。

デフォルト

`-template` オプションを指定しないと、`-template=no%wholeclass,extdef` が使用されます。

例

次のコードについて考えてみましょう。

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

-template=geninlinefuncs を指定した場合、S の 2 つのメンバー関数は、プログラム内で呼び出されなくてもオブジェクトファイルに生成されます。

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                           [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                           [__1cBS4Ci_Dsmf6F_v_]
```

関連項目

86 ページの「全クラスインスタンス化」、 94 ページの「テンプレート定義の検索」

-time

-xtime と同じです。

-U *name*

プリプロセッサシンボル *name* の初期定義を削除します。

このオプションは、コマンド行に指定された (CC ドライバによって暗黙的に挿入されるものも含む) -D オプションによって作成されるマクロシンボル *name* の初期定義を削除します。他の定義済みマクロや、ソースファイル内のマクロ定義が影響を受けることはありません。

CC ドライバにより定義される -D オプションを表示するには、コマンド行に -dryrun オプションを追加します。

例

次のコマンドでは、事前に定義されているシンボル __sun を未定義にします。#ifdef （__sun）のような foo.cc 中のプリプロセッサ分では、シンボルが未定義であると検出されます。

```
example% CC -U__sun foo.cc
```

相互の関連性

コマンド行には複数の -U オプションを指定できます。

すべての -U オプションは、存在している任意の -D オプションの後に処理されます。つまり、同じ *name* がコマンド行上の -D と -U の両方に指定されている場合は、オプションが表示される順序にかかわらず *name* は未定義になります。

関連項目

-D

-unroll=*n*

-xunroll=*n* と同じです。

-V

-verbose=version と同じです。

-v

-verbose=diags と同じです。

-vdelx

互換モード (-compat[=4]) のみ

delete[] を使用する式に対し、実行時ライブラリ関数 _vector_delete_ の呼び出しを生成する代わりに _vector_deletex_ の呼び出しを生成します。関数 _vector_delete_ は、削除するポインタおよび各配列要素のサイズという 2 つの引数をとります。

関数 _vector_deletex_ は _vector_delete_ と同じように動作しますが、3 つ目の引数としてそのクラスのデストラクタのアドレスをとります。この引数はサン以外のベンダーが使用するためのもので、関数では使用しません。

デフォルト

コンパイラは、delete[] を使用する式に対して _vector_delete_ の呼び出しを生成します。

警告

これは旧式フラグであり、将来のリリースでは削除されます。サン以外のベンダーからソフトウェアを購入し、ベンダーがこのフラグの使用を推奨していない限り、このオプションは使用しないでください。

-verbose=*v*[,*v*...]

コンパイラメッセージの詳細度を制御します。

値

v には、次に示す値の 1 つを指定します。

表 A-23 -verbose の値

v の値	意味
[no%]diags	各コンパイル段階が渡すコマンド行を表示します [しません]。
[no%]template	テンプレートインスタンス化冗長モード (検証モードともいう) を起動します [しません]。冗長モードはコンパイル中にインスタンス化の各段階の進行を表示します。

表 A-23 -verbose の値

v の値	意味
[no%]version	CC ドライバに対し、呼び出したプログラムの名前とバージョン番号を表示するよう指示します [しません]。
%all	上のすべてを呼び出します。
%none	-verbose=%none は -verbose=no%template,no%diags,no%version を指定することと同じです。

デフォルト

-verbose を指定しないと、-verbose=%none が使用されます。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

+w

意図しない結果が生じる可能性のあるコードを特定します。+w オプションは、関数が大きすぎてインライン化できない場合、および宣言されたプログラム要素が未使用の場合に警告を生成しません。これらの警告は、ソース中の実際の問題を特定するものではないため、開発環境によっては不適切です。そのような環境では、+w でこれらの警告を生成しないようにすることで、+w をより効果的に使用することができます。これらの警告は、+w2 オプションの場合は生成されます。

次のような問題のありそうな構造について、追加の警告を生成します。

- 移植性がない
- 間違っていると考えられる
- 効率が悪い

デフォルト

このオプションを指定しないと、コンパイラは必ず問題となる構造についてのみ警告を出力します。

相互の関連性

+w を指定してコンパイルすると、一部の C++ 標準ヘッダに関する警告が発行されます。

関連項目

-w, +w2

+w2

+w で発行される警告に加えて、技術的な違反についての警告を発行します。+w2 で行われる警告は、危険性はないが、プログラムの移植性を損なう可能性がある違反に対するものです。

+w2 オプションは、システムのヘッダーファイル中で実装に依存する構造が使用されている場合をレポートしなくなりました。システムヘッダーファイルが実装であるため、これらの警告は不適切でした。+w2 でこれらの警告を生成しないようにすることで、+w2 をより効果的に使用することができます。

警告

+w2 を指定してコンパイルすると、Solaris および C++ 標準ヘッダーファイルに関する警告が発行されることがあります。

関連項目

+w

-w

ほとんどの警告メッセージを抑止します。

コンパイラが出す警告を出力しません。ただし、一部の警告、特に旧式の構文に関する重要な警告は抑制できません。

関連項目

+w

-Xm

-features=iddollar と同じです。

-xa

プロファイル用のコードを生成します。

コンパイル時に TCOVDIR 環境変数を設定すれば、カバレージ (.d) ファイルを置くディレクトリを指定できます。この変数を設定しなければ、カバレージ (.d) ファイルは ソースファイルと同じディレクトリにソースファイルとして残ります。

このオプションは、古いカバレージファイルとの下位互換を保つためだけに使用してください。

相互の関連性

-xprofile=tcov オプションと -xa オプションは、1 つの実行可能ファイルで同時に使用できます。すなわち、-xprofile=tcov でコンパイルされたファイルと -xa でコンパイルされたファイルが両方含まれたプログラムをリンクすることができます。1 つのファイルを両方のオプションでコンパイルすることはできません。

-xa オプションと -g を一緒に使用することはできません。

警告

コンパイルとリンクを別々に行う場合で、-xa でコンパイルした場合は、リンクも -xa で行わなければなりません。そうしないと予期できない結果になることがあります。

関連項目

--xprofile=tcov、tcov(1) のマニュアルページ、『プログラムのパフォーマンス解析』

-xalias_level[=*n*]

(SPARC) C++ コンパイラで次のコマンドを指定して、型に基づく別名の解析および最適化を実行することができます。

■ `-xalias_level[=`*n*`]`

ここで、*n* には `any`、`simple`、`compatible` のいずれかを指定します。

■ `-xalias_level=any`

このレベルの解析では、ある型を別名で定義できるとものとして処理されます。ただしこの場合でも、一部の最適化が可能です。

■ `-xalias_level=simple`

単純型は別名で定義されていないものとして処理されます。以下の単純型のいずれかの動的な型である記憶オブジェクトの場合を説明します。

char	short int	long int	float
signed char	unsigned short int	unsigned long int	double
unsigned char	int	long long int	long double
wchar_t	unsigned int	unsigned long long int	enumeration types
データポインタ型	関数ポインタ型	データメンバーのポインタ型	関数メンバーのポインタ型

これらは、以下の型の lvalue を使用してだけアクセスされます。

- オブジェクトの動的な型
- オブジェクトの動的な型を `constant` または `volatile` で修飾したもの。つまり、オブジェクトの動的な型に相当する符号付きまたは符号なしの型。
- オブジェクトの動的な型を `constant` または `volatile` で修飾したものに相当する、符号付きまたは符号なしの型。
- 前述の型のいずれかがメンバーに含まれる集合体または共用体 (再帰的に、その下位の集合体またはそれに含まれる共用体のメンバーについても該当します)。
- char 型または unsigned char 型

■ `-xalias_level=compatible`

配置非互換の型は、別名で定義されていないものとして処理されます。記憶オブジェクトは、以下の型の lvalue を使用してだけアクセスされます。

- オブジェクトの動的な型
- オブジェクトの動的な型を `constant` または `volatile` で修飾したもの。つまり、オブジェクトの動的な型に相当する符号付きまたは符号なしの型。

- オブジェクトの動的な型を `constant` または `volatile` で修飾したものに相当する、符号付きまたは符号なしの型。
- 前述の型のいずれかがメンバーに含まれる集合体または共用体 (再帰的に、その下位の集合体またはそれに含まれる共用体のメンバーについても該当します)。
- オブジェクトの動的な型の (多くの場合は `constant` または `volatile` で修飾した) 基本クラス型。
- char 型または unsigned char 型

コンパイラでは、すべての参照の型が、相当する記憶オブジェクトの動的な型と配置互換であるものと見なされます。2 つの型は、以下の条件の場合に配置互換になります。

- 2 つの型が同一の型の場合は、配置互換になります。
- 2 つの型の違いが、修飾が constant か volatile かの違いだけの場合は、配置互換になります。
- 符号付き整数型それぞれに、それに相当する (ただしそれとは異なる) 符号なし整数型があります。これらの相当する型は配置互換になります。
- 2 つの列挙型は、基礎の型が同一の場合に配置互換になります。
- 2 つの Plain Old Data (POD) 構造体型は、メンバー数が同一で、順序で対応するメンバーが配置互換である場合に配置互換になります。
- 2 つの POD 共用体型は、メンバー数が同一で、対応するメンバー (順番は任意) が配置互換である場合に配置互換になります。

参照は、一部の場合に、記憶オブジェクトの動的な型と配置非互換になります。

- POD 共用体に、開始シーケンスが共通の POD 構造体が複数含まれていて、そのPOD 共用体オブジェクトにそれらの POD 構造体のいずれかが含まれている場合は、任意の POD 構造体の共通の開始部分を調べることができます。2 つの POD 構造体が共通の開始シーケンスを共有していて、対応するメンバーの型が配置互換であり、開始メンバーのシーケンスでビットフィールドの幅が同一の場合に、2 つの POD 構造体は開始シーケンスが共通になります。
- reinterpret_cast を使用して正しく変換した POD 構造体オブジェクトへのポインタは、その最初のメンバーを示します。そのメンバーがビットフィールドの場合は、そのビットフィールドのあるユニットを示します。

デフォルト

-xalias_level を指定しない場合は、コンパイラでは -xalias_level=any が指定されます。-xalias_level を値なしで指定した場合は、コンパイラでは -xalias_level=compatible が指定されます。

相互の関連性

コンパイラは、-x02 以下の最適化レベルでは、型に基づく別名の解析および最適化を実行しません。

警告:

reinterpret_cast またはこれに相当する旧形式のキャストを使用している場合には、解析の前提にプログラムが違反することがあります。また、次の例にあるような共用体の型のパンニングも、解析の前提に違反します。

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

-xar

アーカイブライブラリを作成します。

テンプレートを使用する C++ のアーカイブをコンパイルするときには通常、テンプレートデータベース中でインスタンス化されたテンプレート関数をそのアーカイブの中にあらかじめ入れておく必要があります。このオプションはそれらのテンプレートを必要に応じてアーカイブに自動的に追加します。

値

-xar を指定すると、ar -c-r が起動され、アーカイブがゼロから作成されます。

例

次のコマンド行は、ライブラリファイルとオブジェクトファイルに含まれるテンプレート関数をアーカイブします。

```
example% CC -xar -o libmain.a a.o b.o c.o
```

警告

テンプレートデータベースの .o ファイルをコマンド行に追加しないでください。

アーカイブを構築するときは、ar コマンドを使用しないでください。CC -xar を使用して、テンプレートのインスタンス化情報が自動的にアーカイブに含まれるようにしてください。

関連項目

ar(1)、第 16 章

-xarch=*isa*

対象となる命令セットアーキテクチャ (ISA) を指定します 。

このオプションは、コンパイラが生成するコードを、指定した命令セットアーキテクチャの命令だけに制限します。このオプションは、すべてのターゲットを対象とするような命令としての使用は保証しません。ただし、このオプションを使用するとバイナリプログラムの移植性に影響を与える可能性があります。

値

SPARC プラットフォームの場合

表 A-24 に、SPARC プラットフォームでの各 -xarch キーワードの詳細を示します。

表 A-24 SPARC プラットフォームでの -xarch の値

isa の値	意味
generic	**大多数のシステムで良好なパフォーマンスを得られるように 32 ビットのオブジェクトバイナリを生成します。**これはデフォルトです。このオプションは、どのプロセッサでも大きくパフォーマンスを落とさず、またほとんどのプロセッサで良好なパフォーマンスを得られるような最良の命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。現在は -xarch=v7 と同じです。
generic64	**大多数の 64 ビットのプラットフォームアーキテクチャーで良好なパフォーマンスを得られるように 64 ビットのオブジェクトバイナリを生成します。**このオプションは、どのプロセッサでも大きくパフォーマンスを落とさず、64 ビットカーネルにより Solaris オペレーティング環境での良好なパフォーマンスを得られるように最良な命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。現在、この値は -xarch=v9 に相当します。
native	**現在のシステムで良好なパフォーマンスを得られるように 32 ビットのオブジェクトバイナリを生成します。**これは -fast オプションのデフォルトです。現在プロセッサを実行しているシステムにもっとも適した設定を選択します。
native64	**現在のシステムで良好なパフォーマンスを得られるように 64 ビットのオブジェクトバイナリを生成します。**コンパイラは現在プロセッサを実行しているシステムにもっとも適した設定を選択します。
v7	**SPARC-V7 ISA 用にコンパイルします。**V7 ISA 上で良好なパフォーマンスを得るためのコードを生成します。これは、V8 ISA 上で最良なパフォーマンスを得るための最良の命令セットと同じですが、整数の mul と div 命令、および fsmuld 命令は含まれていません。 以下に例を示します。SPARCstation 1, SPARCstation 2
v8a	**V8a 版の SPARC-V8 ISA 用にコンパイルします。**定義上、V8a は V8 ISA を意味します。ただし、fsmuld 命令は含まれていません。V8a ISA 上で良好なパフォーマンスを得るためのコードを生成します。 例:microSPARC I チップアーキテクチャに基づくすべてのシステム

表 A-24 **SPARC** プラットフォームでの `-xarch` の値 **(**続き**)**

isa の値	意味
`v8`	**SPARC-V8 ISA 用にコンパイルします。**V8 アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 例:SPARCstation 10
`v8plus`	**V8plus 版の SPARC-V9 ISA 用にコンパイルします。** 定義上、V8plus は V9 ISA を意味します。ただし、V8plus ISA 仕様で定義されている 32 ビットサブセットに限定されます。さらに、VIS (Visual Instruction Set) と実装に固有な ISA 拡張機能は含まれていません。 • V8plus ISA 上で良好なパフォーマンスを得るためのコードを生成します。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例:UltraSPARC チップアーキテクチャに基づくすべてのシステム
`v8plusa`	**V8plusa 版の SPARC-V9 ISA 用にコンパイルします。** 定義上、V8plusa は V8plus アーキテクチャ + VIS (Visual Instruction Set) バージョン 1.0 + UltraSPARC 拡張機能を意味します。 • UltraSPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。ただし、V8plus 仕様で定義されている 32 ビットサブセットに限定されます。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例:UltraSPARC チップアーキテクチャに基づくすべてのシステム
`v8plusb`	**UltraSPARC-III 拡張機能を持つ、V8plusb 版の SPARC-V8plus ISA 用にコンパイルします。**UltraSPARC アーキテクチャ + VIS (Visual Instruction Set) バージョン 2.0 + UltraSPARC-III 拡張機能用のオブジェクトコードを生成します。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式です。このコードは Solaris UltraSPARC-III 環境でのみ実行できます。 • UltraSPARC-III アーキテクチャ上で良好なパフォーマンスを得るための最良のコードを使用します。

表 A-24 **SPARC** プラットフォームでの `-xarch` の値 **(**続き**)**

isa の値	意味
`v9`	**SPARC-V9 ISA 用にコンパイルします。**V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される `.o` オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • `-xarch=v9` は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。
`v9a`	**UltraSPARC 拡張機能を持つ SPARC-V9 ISA 用にコンパイルします。** SPARC-V9 ISA に VIS (Visual Instruction Set) と UltraSPARC プロセッサに固有の拡張機能を追加します。V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される `.o` オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • `-xarch=v9` は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

表 A-24 SPARC プラットフォームでの -xarch の値 (続き)

isa の値	意味
v9b	UltraSPARC-III 拡張機能を持つ SPARC-V9 ISA 用にコンパイルします。 V9a 版の SPARC-V9 ISA に UltraSPARC-III 拡張と VIS バージョン 2.0 を追加します。Solaris UltraSPARC-III 環境で良好なパフォーマンスを得るためのコードを生成します。 • 生成される .o オブジェクトファイルは SPARC-V9 ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • -xarch=v9b は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

また、次のことにも注意してください。

- SPARC 命令セットアーキテクチャ V7、V8 および V8a はバイナリ互換です。
- v8plus でコンパイルされたオブジェクトバイナリ (.o) ファイルとv8plusa でコンパイルされた .o ファイルは、SPARC V8plusa 互換のプラットフォーム上でのみリンクおよび同時に実行できます。
- v8plus、v8plusa、およびv8plusb でそれぞれコンパイルされたオブジェクトバイナリ(.o)ファイル は、SPARC V8plusb 互換のプラットフォーム上でのみリンクおよび同時に実行できます。
- -xarch の値 generic64、native64、v9、v9a および v9b は、UltraSPARC 64 ビット Solaris 環境でのみ指定できます。
- generic64、native64、v9 と v9a でそれぞれコンパイルされたオブジェクトバイナリ (.o) ファイルは、SPARC V9a 互換プラットフォーム上でのみリンクおよび同時に実行できます。
- generic64、native64、v9、v9a、および v9b でそれぞれコンパイルされたオブジェクトバイナリ (.o) ファイルは、SPARC V9b 互換プラットフォーム上でのみリンクおよび同時に実行できます。

いずれの場合でも、初期のアーキテクチャでは、生成された実行可能ファイルの実行速度がかなり遅くなる可能性があります。また、4 倍精度 (REAL*16 と long double) の浮動小数点命令は多くの命令セットアーキテクチャで使用できますが、この命令はコンパイラが使用するコードには含まれません。

IA プラットフォームの場合

表 A-25に、IA プラットフォームでの -xarch キーワードの詳細を示します。

表 A-25 IA プラットフォームでの -xarch の値

isa の値	意味
generic	ほとんどのシステムで良好なパフォーマンスを得られるようにコンパイルします。これはデフォルトです。このオプションは、どのプロセッサでも大きくパフォーマンスを落とさず、またほとんどのプロセッサで良好なパフォーマンスを得られるような最良の命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。
386	このリリースでは、generic と 386 は同じです。
pentium_pro	命令セットを pentium_pro アーキテクチャに限定します。

デフォルト

-xarch=*isa* を指定しないと、-xarc=generic が使用されます。

相互の関連性

このオプションは単体でも使用できますが、-xtarget オプションの展開の一部でもあります。したがって、特定の -xtarget オプションによって展開された -xarch の値を変更するためにも使用できます。たとえば、-xtarget=ultra2 は -xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1 に展開されます。次のコマンドでは、-xarch=v8plusb は、-xtarget=ultra2 の展開で設定された -xarch=v8plusa より優先されます。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

-compat[=4] とともに -xarch=generic64、-xarch=native64、-xarch=v9、-xarch=v9a、-xarch=v9b のいずれかを使用することはできません。

警告

このオプションを最適化と併せて使用する場合、適切なアーキテクチャを選択すると、そのアーキテクチャ上での実行パフォーマンスを向上させることができます。ただし、適切な選択をしなかった場合、パフォーマンスが著しく低下するか、あるいは、作成されたバイナリプログラムが目的のターゲットプラットフォーム上で実行できない可能性があります。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ `-xarch` の値を指定してください。

-xbuiltin[={%all|%none}]

標準ライブラリ呼び出しの最適化を有効または無効にします。

デフォルトでは、標準ライブラリヘッダで宣言された関数は、コンパイラによって通常の関数として処理されます。ただし、これらの関数の一部は、「組み込み」として認識されます。組み込み関数として処理されるときは、コンパイラでより効果的なコードを生成できます。たとえば、一部の関数は副作用がないことをコンパイラで認識でき、同じ入力が与えられると常に同じ出力が戻されます。一部の関数はコンパイラによって直接インラインで生成できます。

`-xbuiltin=%all` オプションは、コンパイラにできるだけ多数の組み込み標準関数を認識するように指示します。認識される関数の正確なリストは、コンパイラコードジェネレータのバージョンによって異なります。

`-xbuiltin=%none` オプションはデフォルトのコンパイラの動作に影響を与え、コンパイラは組み込み関数に対して特別な最適化は行いません。

デフォルト

`-xbuiltin` を指定しないと、コンパイラでは `-xbuiltin=%none` が使用されます。

`-xbuiltin` だけを指定すると、コンパイラでは `-xbuiltin=%all` が使用されます。

相互の関連性

マクロ `-fast` の拡張には、`-xbuiltin=%all` が取り込まれます。

例

次のコンパイラコマンドでは、標準ライブラリ呼び出しを特殊処理するように要求します。

```
example% CC -xbuiltin -c foo.cc
```

次のコンパイラコマンドでは、標準ライブラリ呼び出しを特殊処理しないように要求します。マクロ -fast の拡張には -xbuiltin=%all が取り込まれていることに注意してください。

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

-xcache=*c*

SPARC:オプティマイザ用のキャッシュ特性を定義します。

オプティマイザが使用できるキャッシュの属性を定義します。この定義によって、特定のキャッシュが使用されるわけではありません。

注 – このオプションは単独でも使用できますが、-xtarget オプションが展開されたものの一部です。このオプションの主な目的は、-xtarget オプションにより指定される値を変更することです。

値

c には次の値のいずれかを指定します。

表 A-26 -xcache の値

c の値	意味
generic	これはデフォルトです。ほとんどの SPARC プロセッサに良好なパフォーマンスを提供し、どの x86、SPARC の各プロセッサでも著しいパフォーマンス低下が生じないようなキャッシュ特性を使用するように、コンパイラに指示します。 これらの最高のタイミング特性は、新しいリリースごとに、必要に応じて調整されます。
native	ホスト環境に対して最適化されたパラメータを設定します。
s1/l1/a1	レベル 1 のキャッシュ属性を定義します。
s1/l1/a1:s2/l2/a2	レベル 1 とレベル 2 のキャッシュ属性を定義します。
s1/l1/a1:s2/l2/a2:s3/l3/a3	レベル 1、レベル 2、レベル 3 のキャッシュ属性を定義します。

キャッシュ属性 *si/li/ai,* の定義は次のとおりです。

属性	定義
si	レベル *i* のデータキャッシュのサイズ (K バイト)
li	レベル *i* のデータキャッシュのラインサイズ (バイト)
ai	レベル *i* のデータキャッシュの結合規則

たとえば、*i*=1 は、レベル 1 のキャッシュ属性の *s1/l1/a1* を味します。

デフォルト

-xcache を指定しないと、-xcache=generic がデフォルトで使用されます。この値を指定すると、ほとんどの SPARC プロセッサで良好なパフォーマンスが得られ、どのプロセッサでも顕著なパフォーマンスの低下がないキャッシュ属性がコンパイラで使用されます。

例

`-xcache=16/32/4:1024/32/1` では、以下の内容を指定します。

レベル 1 のキャッシュ	レベル 2 のキャッシュ
16K バイト	1024K バイト
32 バイトの行サイズ	32 バイトの行サイズ
4 ウェイアソシアティブ	直接マップ結合

関連項目

`-xtarget=t`

-xcg89

`-xtarget=ss2` と同じです。

警告

コンパイルとリンクを別々に実行する場合で、コンパイルで `-xcg89` を使用した場合は、リンクでも同じオプションを使用してください。そうしないと、予期しない結果が発生する可能性があります。

-xcg92

`-xtarget=ss1000` と同じです。

警告

コンパイルとリンクを別々に実行する場合で、コンパイルで `-xcg92` を使用した場合は、リンクでも同じオプションを使用してください。そうしないと、予期しない結果が発生する可能性があります。

-xchar[=*o*]

このオプションは、char 型が符号なしで定義されているシステムからのコード移植を簡単にするためのものです。そのようなシステムからの移植以外では、このオプションは使用しないでください。符号付きまたは符号なしであると明示的に書き直す必要があるのは、符号に依存するコードだけです。

値

o には、以下のいずれかを指定します。

表 A-27 -xchar の値

値	意味
signed	char 型で定義された文字定数および変数を符号付きとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
s	signed と同義です。
unsigned	char 型で定義された文字定数および変数を符号なしとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
u	unsigned と同義です。

デフォルト

-xchar を指定しない場合は、コンパイラでは -xchar=s が指定されます。

-xchar を値なしで指定した場合は、コンパイラでは -xchar=s が指定されます。

相互の関連性

-xchar オプションは、-xchar を指定してコンパイルしたコードでだけ、char 型の値の範囲を変更します。システムルーチンまたはヘッダーファイル内の char 型の値の範囲は変更しません。特に、CHAR_MAX および CHAR_MIN の値 (limits.h で定義される) は、このオプションを指定しても変更されません。したがって、CHAR_MAX および CHAR_MIN は、通常の char で符号化可能な値の範囲を示さなくなります。

警告

-xchar を使用するときは、マクロでの値が符号付きの場合があるため、char を定義済みのシステムマクロと比較する際には特に注意してください。これは、マクロを使用してエラーコードを戻すルーチンでもっとも一般的です。エラーコードは、一般的には負の値になっています。したがって、char をそのようなマクロによる値と比較するときは、結果は常に false になります。負の数値が符号なしの型の値と等しくなることはありません。

ライブラリを使用してエクスポートしているインタフェース用のルーチンは、-xchar を使用してコンパイルしないようにお勧めします。Solaris ABI では char 型を符号付きと指定しており、それに従ってシステムライブラリが動作します。char 型を符号なしにする影響は、システムライブラリを使って幅広くテストされてはいません。このオプションを使用する代わりに、char 型の符号の有無に依存しないようにコードを変更してください。char 型の符号は、コンパイラやオペレーティングシステムによって異なります。

-xcheck[=*i*]

SPARC:-xcheck=stkovf を指定してコンパイルすると、シングルスレッドのプログラム内のメインスレッドのスタックオーバーフローおよびマルチスレッドプログラム内のスレーブスレッドのスタックが実行時にチェックされます。スタックオーバーフローが検出された場合は、SIGSEGV が生成されます。アプリケーションで、スタックオーバーフローで生成される SIGSEGV を他のアドレス空間違反と異なる方法で処理する必要がある場合は、sigaltstack(2) を参照してください。

値

i には、以下のいずれかを指定します。

表 A-28 -xcheck の値

値	意味
%all	チェックをすべて実行します。
%none	チェックを実行しません。
stkovf	スタックオーバーフローのチェックをオンにします。
no%stkovf	スタックオーバーフローのチェックをオフにします。

デフォルト

-xcheck を指定しない場合は、コンパイラではデフォルトで -xcheck=%none が指定されます。

引数を指定せずに -xcheck を使用した場合は、コンパイラではデフォルトで -xcheck=%none が指定されます。

-xcheck オプションは、コマンド行で累積されません。コンパイラは、コマンドで最後に指定したものに従ってフラグを設定します。

-xchip=*c*

オプティマイザが使用するターゲットとなるプロセッサを指定します。

ターゲットとなるプロセッサを指定することによって、タイミング属性を指定します。このオプションは次のものに影響を与えます。

- 命令の順序 (スケジューリング)
- コンパイラが分岐を使用する方法
- 意味が同じもので代用できる場合に使用する命令

注 – このオプションは単独でも使用できますが、-xtarget オプションが展開されたものの一部です。このオプションの主な目的は、-xtarget オプションにより指定される値を変更することです。

値

c には次の値のいずれかを指定します。

表 A-29 `-xchip` の値

プラットフォーム	*c* の値	タイミング属性を使用する意味
SPARC	`generic`	SPARC プロセッサ上で良好なパフォーマンスを得るための、タイミング属性プロセッサ
	`native`	現在コンパイルを実行しているシステム上で良好なパフォーマンスを得るため
	`old`	SuperSPARC プロセッサより古いプロセッサのタイミング属性
	`super`	SuperSPARC プロセッサのタイミング属性
	`super2`	SuperSPARC II プロセッサのタイミング属性
	`micro`	MicroSPARC プロセッサのタイミング属性
	`micro2`	MicroSPARC II プロセッサのタイミング属性
	`hyper`	HyperSPARC プロセッサのタイミング属性
	`hyper2`	HyperSPARC II プロセッサのタイミング属性
	`powerup`	Weitek PowerUp プロセッサのタイミング属性
	`ultra`	UltraSPARC I プロセッサのタイミング属性
	`ultra2`	UltraSPARC II プロセッサのタイミング属性
	`ultra2e`	UltraSPARC IIe プロセッサのタイミング属性
	`ultra2i`	UltraSPARC IIi プロセッサのタイミング属性
	`ultra3`	UltraSPARC III プロセッサのタイミング属性
	`ultra3cu`	UltraSPARC III Cu プロセッサのタイミング属性
IA	`generic`	一般的な IA プロセッサが持つタイミング属性プロセッサ
	`386`	Intel 386 プロセッサのタイミング属性
	`486`	Intel 486 プロセッサのタイミング属性
	`pentium`	Intel Pentium プロセッサのタイミング属性
	`pentium_pro`	Intel Pentium Pro チップのタイミング属性

デフォルト

ほとんどの SPARC プロセッサでは、デフォルト値の `generic` を使用すれば、どのプロセッサでもパフォーマンスの著しい低下がなく、良好なパフォーマンスが得られる最良のタイミング属性がコンパイラで使用されます。

`-xcode=`*a*

SPARC:コードのアドレス空間を指定します。

注 – `-xcode=pic13` または `-xcode=pic32` を指定して共有オブジェクトを構築することをお勧めします。`-xarch=v9 -xcode=abs64` と `-xarch=v8 -xcode=abs32` を指定して有効な共有オブジェクトを構築することは可能ですが、効率は悪くなります。`-xarch=v9 -xcode=abs32` または `-xarch=v9 -xcode=abs44` を指定して構築したオブジェクトは機能しません。

値

a には次のいずれかを指定します。

表 A-30 `-xcode` の値

a の値	意味
`abs32`	32 ビット絶対アドレスを生成します。高速ですが範囲が限定されます。コード、データ、および bss を合計したサイズは 2**32 バイトに制限されます。
`abs44`	SPARC:44 ビット絶対アドレスを生成します。中程度の速さで中程度の範囲を使用できます。コード、データ、および bss を合計したサイズは 2**44 バイトに制限されます。64 ビットのアーキテクチャでのみ利用できます。 `-xarch={v9|v9a|v9b}`

表 A-30 -xcode の値 (続き)

a の値	意味
abs64	SPARC:64 ビット絶対アドレスを生成します。低速ですが範囲は制限されません。64 ビットのアーキテクチャでのみ利用できます。 -xarch={v9|v9a|v9|generic64|native64}
pic13	位置に依存しないコード (小規模モデル) を生成します。高速ですが範囲が限定されます。-Kpic と同じです。32 ビットアーキテクチャでは最大 2**11 個の固有の外部シンボルを、64 ビットでは 2**10 個の固有の外部シンボルをそれぞれ参照できます。
pic32	位置に依存しないコード (大規模モデル) を生成します。低速ですが全範囲を使用できます。-KPIC と同じです。32 ビットアーキテクチャでは最大 2**30 個の固有の外部シンボルを、64 ビットでは 2**29 個の固有の外部シンボルをそれぞれ参照できます。

デフォルト

SPARC V8 と V7 の場合は -xcode=abs32 です。

SPARC と UltraSPARC (-xarc={v9|v9a|v9b|generic64|native64} のとき) の場合は -xcode=abs64 です。

SPARC の場合、-xcode=pic13 および -xcode=pic32 では、わずかですが、次の 2 つのパフォーマンス上の負担がかかります。

- -xcode=pic13 および -xcode=pic32 のいずれかでコンパイルしたルーチンは、共有ライブラリの大域または静的変数へのアクセスに使用されるテーブル (_GLOBAL_OFFSET_TABLE_) を指し示すようレジスタを設定するために、入口で命令を数個余計に実行します。
- 大域または静的変数へのアクセスのたびに、_GLOBAL_OFFSET_TABLE_ を使用した間接メモリー参照が 1 回余計に行われます。-xcode=pic32 でコンパイルした場合は、大域および静的変数への参照ごとに命令が 2 個増えます。

こうした負担があるとしても、-xcode=pic13 あるいは -xcode=pic32 を使用すると、ライブラリコードを共有できるため、必要となるシステムメモリーを大幅に減らすことができます。-xcode=pic13 あるいは -xcode=pic32 でコンパイルした共有ライブラリを使用するすべてのプロセスは、そのライブラリのすべてのコードを共有できます。共有ライブラリ内のコードに非 pic (すなわち、絶対) メモリー参照が 1

つでも含まれている場合、そのコードは共有不可になるため、そのライブラリを使用するプログラムを実行する場合は、その都度、コードのコピーを作成する必要があります。

.o ファイルが -xcode=pic13 または -xcode=pic32 でコンパイルされたかどうかを調べるには、次のように nm コマンドを使用すると便利です。

```
% nm ファイル名.o | grep _GLOBAL_OFFSET_TABLE_ U
_GLOBAL_OFFSET_TABLE_
```

位置独立コードを含む .o ファイルには、_GLOBAL_OFFSET_TABLE_ への未解決の外部参照が含まれます。このことは、英字の U で示されます。

-xcode=pic13 または -xcode=pic32 を使用すべきかどうかを判断するには、nm を使用して、共有ライブラリで使用または定義されている明確な大域および静的変数の個数を確認します。_GLOBAL_OFFSET_TABLE_ のサイズが 8,192 バイトより小さい場合は、-Kpic を使用できます。そうでない場合は、-xcode=pic32 を使用する必要があります。

警告

別々の手順でコンパイルしてリンクする場合は、コンパイル手順とリンク手順で同じ -xarch オプションを使用する必要があります。

-xcrossfile[=*n*]

SPARC:複数ソースファイルに渡る最適化とインライン化を有効にする-xcrossfile は、コンパイル時に機能し、コンパイルコマンドで指定したファイルだけに対して有効になります。次にコマンド行の例を示します。

```
example% CC -xcrossfile -xO4 -c f1.cc f2.cc
example% CC -xcrossfile -xO4 -c f3.cc f4.cc
```

f1.cc ファイルと f2.cc ファイルの間、および f3.cc ファイルとf4.cc ファイルの間でクロスモジュールの最適化が行われます。f1.cc と f3.cc または f1.cc と f4.cc の間では最適化は行われません。

値

a には次の値のいずれかを指定します。

表 A-31 `-xcrossfile` の値

n の値	意味
`0`	複数のソースファイルに渡る最適化とインライン化を実行しません。
`1`	複数のソースファイルに渡る最適化とインライン化を実行します。

通常、コンパイラの解析の範囲は、コマンド行で指定した個々のファイルごとに行われます。たとえば、`-xO4` オプションを指定した場合、自動インライン化は同じソースファイル内で定義および参照されているサブプログラムにのみ行われます。

`-xcrossfile` または `-xcrossfile=1` を指定すると、コンパイラはコマンド行で指定されたすべてのファイルを一括して分析し、それらが単一のソースファイルであるかのように扱います。

デフォルト

`-xcrossfile` を指定しない場合、`-xcrossfile=0` が仮定され、複数のソースファイルに渡る最適化とインライン化は行われません。

`-xcrossfile` は`-xcrossfile=1` と同じです。

相互の関連性

`-xcrossfile` オプションは、`-xO4` または`-xO5` と一緒に使用した場合にのみ効果が得られます。

警告

このオプションを使ってコンパイルされたファイルは、インライン化されたコードを含む可能性があるため、相互に依存しています。したがって、プログラムにリンクするときは、1 つの単位として使用しなければなりません。あるルーチンを変更したた

めに、関連するファイルを再コンパイルした場合は、すべてのファイルを再コンパイルする必要があります。結果として、このオプションを使用すると、makefile の構成に影響を与えます。

関連項目

-xldscope

-xdumpmacros[=*value[*, *value...*]]

マクロがプログラム内でどのように動作しているかを調べたいときに、このオプションを使用します。このオプションは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に従って、標準エラー (stderr) に出力されます。-xdumpmacros オプションは、ファイルの終わりまで、または dumpmacros プラグマまたは end_dumpmacros プラグマによって上書きされるまで有効です。426 ページの「#pragma dumpmacros」を参照してください。

値

value の代わりに次の引数を使用できます。

表 A-32 -xdumpmacros の値

値	意味
[no%]defs	すべての定義済みマクロを出力します [しません]。
[no%]undefs	すべての解除済みマクロを出力します [しません]。
[no%]use	使用されているマクロの情報を出力します [しません]。
[no%]loc	defs、undefs、use の位置 (パス名と行番号) を印刷します [しません]。
[no%]conds	条件付き指令で使用したマクロの使用情報を出力します [しません]。

表 A-32 -xdumpmacros の値 (続き)

値	意味
[no%]sys	システムヘッダーファイルのマクロについて、すべての定義済みマクロ、解除済みマクロ、使用情報を出力します [しません]。
%all	オプションを -xdumpmacros=defs,undefs,use,loc,conds,sys に設定します。この引数は、[no%] 形式の引数と併用すると効果的です。たとえば -xdumpmacros=%all,no%sys は、出力からシステムヘッダーマクロを除外しますが、その他のマクロに関する情報は依然として出力します。
%none	あらゆるマクロ情報を出力しません。

オプションの値は追加されていきます。-xdumpmacros=sys -xdumpmacros=undefs を指定した場合と、-xdumpmacros=undefs,sys. を指定した場合の効果は同じです。

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果を持ちません。たとえば -xdumpmacros=loc,conds,sys は、まったく効果を持ちません。

デフォルト

引数を付けないで -xdumpmacros を指定した場合、-xdumpmacros=defs,undefs,sys を指定したことになります。-xdumpmacros を指定しなかった場合、デフォルト値として -xdumpmacros=%none が使用されます。

例

-xdumpmacros=use,no%loc オプションを使用すると、使用した各マクロの名前が一度だけ印刷されます。より詳しい情報が必要であれば、-xdumpmacros=use,loc オプションを使用します。マクロを使用するたびに、そのマクロの名前と位置が印刷されます。

以下は、ファイル t.c の例です。

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b)  a+b
#else
#define COMPUTE(a,b)  a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

以下の例は、defs、undefs、sys、および loc の引数に基づいた、ファイル t.c の出力を示しています。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_7 1
#define __SUNPRO_CC 0x550
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_7 1
command line: #define __SUNPRO_CC 0x550
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #undef BAR
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

次の例では、use、loc、および conds の引数によって、マクロ動作がファイル t.c に出力されます。

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

以下は、ファイル y.c の例です。

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

以下は、y.c 内のマクロに基づいた -xdumpmacros=use,loc の出力です。

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

関連項目

dumpmacros プラグマと end_dumpmacros プラグマを使用すれば、-xdumpmacros のスコープを変更できます。

-xe

構文エラーと意味エラーの有無チェックのみ行います。-xe を指定すると、オブジェクトコードは出力されません。-xe の出力は、stderr に送られます。

コンパイルによってオブジェクトファイルを生成する必要がない場合には、-xe オプションを使用してください。たとえば、コードの一部を削除することによってエラーメッセージの原因を切り分ける場合には、-xe を使用することによって編集とコンパイルを高速化できます。

関連項目

-c

-xF[=*v*[,*v*...]]

リンカーによる関数と変数の最適な順序の並べ替えを有効にします。

このオプションは、関数とデータ変数を細分化された別々のセクションに配置するようコンパイラに指示します。それによってリンカーは、リンカーの -M オプションで指定されたマップファイル内の指示に従ってこれらのセクションの順序を並べ替えて、プログラムのパフォーマンスを最適化することができます。通常この最適化は、ページフォルト時間が、プログラム実行時間の大半を占める場合にのみ効果があります。

変数を並べ替えることによって、実行時間のパフォーマンスを低下させる次のような問題を解決できます。

- 関連性のない複数の変数がメモリー内で近接することによって引き起こされる、キャッシュとページの競合
- 関連性のある複数の変数が、メモリー内で近接していないことによってもたらされる、不必要に大きい作業セットのサイズ
- 効果的なデータ密度を低下させる弱い変数の使用されないコピーによってもたらされる、不必要に大きい作業セットのサイズ

最適なパフォーマンスを得るために変数と関数の順序を並べ替えるには、次の処理が必要です。

1. `-xF` でコンパイル、リンクします
2. 『プログラムのパフォーマンス解析』マニュアル内の、関数のマップファイルを生成する方法に関する指示に従うか、または、『リンカーとライブラリ』内の、データのマップファイルを生成する方法に関する指示に従います。
3. リンカーの `-M` オプションを使用して新しいマップファイルを再リンクします。
4. アナライザで再実行して、パフォーマンスが向上したかどうかを検証します。

値

v には、以下のいずれかを指定します。

表 A-33 `-xF` の値

値	意味
[no%]func	関数を個々のセクションに細分化します [しません]。
[no%]gbldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
[no%]lcldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
%all	関数、大域データ、局所データを細分化します。
%none	何も細分化しません。

デフォルト

`-xF` を指定しない場合のデフォルトは、`-xF=%none` です。引数を指定しないで `-xF` を指定した場合のデフォルトは、`-xF=%none,func` です。

相互の関連性

`-xF=lcldata` を指定するとアドレス計算最適化が一部禁止されるので、このフラグは実験として意味があるときにだけ使用するとよいでしょう。

関連項目

analyzer(1)、debugger(1)、および ld(1) のマニュアルページ

-xhelp=flags

各コンパイラオプションの簡単な説明を表示します。

-xhelp=readme

README (最新情報) ファイルの内容を表示します。

README ファイルのページングには、環境変数 PAGER で指定されているコマンドが使用されます。PAGER が設定されていない場合、デフォルトのページングコマンド more が使用されます。

-xia

SPARC:区間演算ライブラリをリンクし、適切な浮動小数点環境を設定します。

注 – C++ 区間演算ライブラリは、Fortran コンパイラで実装されているとおり、区間演算と互換性があります。

拡張

-xia オプションは、-fsimple=0 -ftrap=%none -fns=no -library=interval に拡張するマクロです。

相互の関連性

区間演算ライブラリを使用するには、<suninterval.h> を取り込みます。

区間演算ライブラリを使用するときは、libC、Cstd、または iostreams のいずれかのライブラリを取り込む必要があります。これらのライブラリを取り込む方法については、-library を参照してください。

警告

区間を使用し、-fsimple、-ftrap、または -fns にそれぞれ異なる値を指定すると、プログラムの動作が不正確になる可能性があります。

C++ 区間演算は実験に基づくもので発展性があります。詳細はリリースごとに変更される可能性があります。

関連項目

『C++ Interval Arithmetic Programming Reference』、『Interval Arithmetic Solves Nonlinear Problems While Providing Guaranteed Results』(http://www.sun.com/forte/info/features/intervals.html)、-library。

-xildoff

インクリメンタルリンカーを無効にします。

デフォルト

-g オプションを使用していない場合は、この -xildoff オプションがデフォルトになります。さらに -G オプションを使用しているか、コマンド行にソースファイルを指定している場合も、このオプションがデフォルトになります。このオプションを無効にするには、-xildon オプションを使用してください。

関連項目

-xildon 、ild(1) および ld(1) のマニュアルページ、『C ユーザーズガイド』の「インクリメンタルリンカー」

-xildon

インクリメンタルリンカーを有効にします。

-G ではなく -g を使用し、コマンド行にソースファイルを指定していない場合は、このオプションが有効になります。このオプションを無効にするには、-xildoff オプションを使用してください。

関連項目

-xildoff 、ild(1) および ld(1) のマニュアルページ、『C ユーザーズガイド』の「インクリメンタルリンカー」

-xinline[=*func_spec*[,*func_spec*...]]

どのユーザー作成ルーチンをオプティマイザによって -x03 レベル以上でインライン化するかを指定します。

値

func_spec には次の値のいずれかを指定します。

表 A-34 -xinline の値

func_spec の値	意味
%auto	最適化レベル -x04 以上で自動インライン化を有効にします。この引数は、オプティマイザが選択した関数をインライン化できることをオプティマイザに知らせます。%auto の指定がないと、明示的インライン化が-xinline=[no%]*func_name*... によってコマンド行に指定されていると、自動インライン化は通常オフになります。
func_name	オプティマイザに関数をインライン化するように強く要求します。関数が extern "C" で宣言されていない場合は、*func_name* の値を符号化する必要があります。実行可能ファイルに対し nm コマンドを使用して符号化された関数名を検索できます。extern "C" で宣言された関数の場合は、名前はコンパイラで符号化されません。
no%*func_name*	リスト上のルーチン名の前に no% を付けると、そのルーチンのインライン化が禁止されます。*func_name* の符号化名に関する規則は、no%*func_name* にも適用されます。

-xcrossfile[=1] を使用しない限り、コンパイルされているファイルのルーチンだけがインライン化の対象とみなされます。オプティマイザでは、どのルーチンがインライン化に適しているかを判断します。

デフォルト

-xinline オプションを指定しないと、コンパイラでは -xinline=%auto が使用されます。

-xinline= に引数を指定しないと、最適化のレベルにかかわらず関数がインライン化されます。

例

int foo() を宣言している関数のインライン化を無効にして自動インライン化を有効にするには次のコマンドを使用します。

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

int foo() として宣言した関数のインライン化を強く要求し、他のすべての関数をインライン化の候補にするには次のコマンドを使用します。

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

int foo() として宣言した関数のインライン化を強く要求し、その他の関数のインライン化を禁止するには次のコマンドを使用します。

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

相互の関連性

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 以上では、-xinline オプションを指定しなくてもオプティマイザでどの関数をインライン化する必要があるかを判断します。-xO4 では、コンパイラはどの関数が、インライン化されたときにパフォーマンスを改善するかを判断しようとします。

ルーチンは、次のいずれかの条件が当てはまる場合はインライン化されません。警告は必ず出されます。

- 最適化が -xO3 未満
- ルーチンを検出できない

- インライン化が収益性が低く安全性に欠ける
- ソースがコンパイルされているファイルにない、または -xcrossfile[=1] を使用している場合にソースがコマンド行で指定された名前のファイルにない

警告

-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

関連項目

-xldscope

-xipo[={0|1|2}]

内部手続きの最適化を実行します。

-xipo オプションが内部手続きの解析パスを呼び出すことで全プログラムの最適化を実行します。-xcrossfile とは違って、-xipo はリンク手順でのすべてのオブジェクトファイル間の最適化を行い、しかもこれらの最適化は単にコンパイルコマンドのソースファイルにとどまりません。

-xipo オプションは、大量のファイルを使用してアプリケーションをコンパイルしてリンクするときに特に便利です。このフラグを指定してコンパイルされたオブジェクトファイルには、ソースプログラムファイルとコンパイル済みプログラムファイル間で内部手続きの解析を有効にする解析情報が含まれています。ただし、解析と最適化は -xipo を指定してコンパイルされたオブジェクトファイルに限定され、ライブラリのオブジェクトファイルには拡張されません。

値

-xipo オプションには以下の値があります。

表 A-35 -xipo の値

値	意味
0	内部手続きの最適化を実行しません
1	内部手続きの最適化を実行します
2	内部手続きの別名解析と、メモリーの割り当ておよび配置の最適化を実行し、キャッシュ性能を向上します

デフォルト

-xipo を指定しないと、-xipo=0 が使用されます。

-xipo だけを指定すると、-xipo=1 が使用されます。

例

次の例では同じ手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

オプティマイザは 3 つのすべてのソースファイル間でファイル間のインライン化を実行します。ソースファイルのコンパイルをすべて 1 回のコンパイルで実行しないで済むように、またいくつかの個別のコンパイル時にそれぞれ -xipo オプションを指定して行えるように最後のリンク手順でファイル間のインライン化を実行します。

次の例では別々の手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog  part1.o part2.o part3.o
```

このコンパイル手順で作成されたオブジェクトファイルには、ファイル間の最適化がリンク手順で行われるように補足解析情報がコンパイルされています。

相互の関連性

-xipo オプションでは最低でも最適化レベル -x04 が必要です。

同じコンパイラコマンド行に -xipo オプションと -xcrossfile オプションの両方は使用できません。

警告

別々の手順でコンパイルしてリンクする場合は、有効にするために両方の手順に同じ -xipo を指定する必要があります。

-xipo を指定しないでコンパイルされたオブジェクトは、-xipo を指定してコンパイルされたオブジェクトと自由にリンクできます。

ライブラリでは、次の例に示すように、-xipo を指定してコンパイルしている場合でもファイル間の内部手続き解析に関与しません。

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

この例では、内部手続きの最適化は one.cc、two.cc および three.cc 間と main.cc と four.cc 間で実行されますが、main.cc または cour.cc と mylib.a のルーチン間では実行されません (最初のコンパイルは未定義のシンボルに関する警告を生成する場合がありますが、内部手続きの最適化は、コンパイル手順でありしかもリンク手順であるために実行されます) 。

-xipo オプションを指定すると、ファイル間で最適化を行うために必要な補足情報のために極端に大きいオブジェクトファイルが生成されます。ただし、この補足情報は最終的な実行可能バイナリファイルの一部にはなりません。実行可能プログラムのサイズの増加は、その他に最適化を実行したことに起因します。

関連項目

-xjobs

-xjobs=*n*

コンパイラが処理を行うために生成するプロセスの数を設定するには、-xjobs オプションを指定します。このオプションにより、複数の CPU が搭載されているマシンでの構築時間を短縮できます。現在、-xjobs が動作するのは、-xipo オプションを指定した場合だけです。-xjobs=*n* を指定すると、内部手続きオプティマイザは、さまざまなファイルをコンパイルするために呼び出せるコードジェネレータインスタンスの最大数として、*n* を使用します。

値

-xjobs には必ず値を指定する必要があります。値を指定しないと、エラー診断が表示され、コンパイルは中止します。

一般に、*n* に指定する確実な値は、使用できるプロセッサ数に 1.5 を掛けた数です。生成されたジョブ間のコンテキスト切り替えにより生じるオーバーヘッドのため、使用できるプロセッサ数の何倍もの値を指定すると、パフォーマンスが低下することがあります。また、あまり大きな数を使用すると、スワップ領域などシステムリソースの限界を超える場合があります。

デフォルト

コマンド行に複数の -xjobs のインスタンスがある場合、一番右以外の指定は無効になります。

例

次の例に示すコマンドは、2 つのプロセッサを持つシステム上で、-xjobs オプションを指定しないで実行された同じコマンドよりも早くコンパイルを実行します。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

-xlang=*language*[, *language*]

該当する実行時ライブラリをインクルードし、指定された言語に適切な実行時環境を用意する。

値

language は `f77`、`f90`、または `f95` のいずれかとします。

`f90` 引数と `f95` 引数は同じです。

相互の関連性

`-xlang=f90` と `-xlang=f95` の各オプションは `-library=f90` を意味し、`-xlang=f77` オプションは `-library=f77` を意味します。ただし、`-library=f77` と `-library=f90` の各オプションは、`-xlang` オプションしか正しい実行時環境を保証しないので、言語が混合したリンクには不十分です。

言語が混合したリンクの場合、ドライバは次の順序で言語階層を使用してください。

1. C++
2. Fortran 95 (または Fortran 90)
3. Fortran 77
4. C または C99

Fortran 95、Fortran 77、および C++ のオブジェクトファイルを一緒にリンクする場合は、最上位言語のドライバを使用します。たとえば、C++ と Fortran 95 のオブジェクトファイルをリンクするには、次の C++ コンパイラコマンドを使用してください。

```
example% CC -xlang=f95 ...
```

Fortran 95 と Fortran 77 のオブジェクトファイルをリンクするには、次のように Fortran 95 のドライバを使用します。

```
example% f95 -xlang=f77 ...
```

`-xlang` オプションと `-xlic_lib` オプションを同じコンパイラコマンドで使用することはできません。`-xlang` を使用していて、しかも Sun Performance Library でリンクする必要がある場合は、代わりに `-library=sunperf` を使用してください。

警告

`-xlang` と一緒に `-xnolib` を使用しないでください。

Fortran 並列オブジェクトを C++ オブジェクトと混合している場合は、リンク行に -mt フラグを指定する必要があります。

関連項目

-library,-staticlib

-xldscope={*v*}

extern シンボルの定義に対するデフォルトのリンカースコープを変更するには、-xldscope オプションを指定します。デフォルトを変更すると、実装がよりうまく隠蔽されるため、共有ライブラリと実行可能ファイルをより高速かつ安全に使用できるようになります。

値

v には、以下のいずれかを指定します。

表 A-36 -xldscope の値

値	意味
__global	シンボル定義には大域リンカースコープとなります。これは、もっとも限定的でないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープが、外部シンボルのデフォルトのリンカースコープです。
__symbolic	シンボル定義は、シンボリックリンカースコープとなります。これは、大域リンカースコープより限定的なリンカースコープです。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュール外については、シンボルは大域なものとなります。このリンカースコープは、リンカーオプション -Bsymbolic に対応しています。C++ ライブラリでは -Bsymbolic を使用できませんが、__symbolic 指定子は問題なく使用できます。リンカーの詳細については、ld(1) を参照してください。
__hidden	シンボル定義は、隠蔽リンカースコープとなります。隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも限定的なリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。モジュールの外からは、シンボルは見えません。

デフォルト

-xldscope を指定しない場合は、コンパイラでは -xldscope=global が指定されます。値を指定しないで -xldscope を指定すると、コンパイラがエラーを出力します。コマンド行にこのオプションの複数のインスタンスがある場合、一番右にあるインスタンスが実行されるまで前のインスタンスが上書きされていきます。

警告:

クライアントがライブラリ内の関数をオーバーライドすることを許可しようとする場合、ライブラリの構築時に関数がインラインで生成されないようにする必要があります。-xinline を指定して関数名を指定した場合、-xO4 以上でコンパイルした場合(自動的にインライン化されます)、インライン指定を使用した場合、または、複数のソースファイルにわたる最適化を使用している場合、コンパイラは関数をインライン化します。

たとえば、ABC というライブラリにデフォルトのアロケータ関数があり、ライブラリクライアントがその関数を使用でき、ライブラリの内部でも使用されるものとします。

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

-xO4 以上でライブラリを構築すると、コンパイラはライブラリ構成要素内での ABC_allocator の呼び出しをインライン化します。ライブラリクライアントが ABC_allocator を独自のアロケータで置き換える場合、ライブラリ構成要素内では置き換えは行われません。最終的なプログラムには、この関数の相異なるバージョンが含まれることになります。

__hidden または __symbolic 指定子で宣言されたライブラリ関数は、ライブラリの構築時にインラインで生成される可能性があります。これらの関数がクライアントからオーバーライドされることは、サポートされていません。50 ページの「リンカースコープ」を参照してください。

__global 指示子で宣言されたライブラリ関数はインラインで宣言しないでください。また、-xinline コンパイラオプションを使用することによってインライン化されることがないようにしてください。

関連項目

-xinline、-xO、xcrossfile

-xlibmieee

例外時に libm が数学ルーチンに対し IEEE 754 の値を返す。

libm のデフォルト動作は XPG に準拠します。

関連項目

『数値計算ガイド』

-xlibmil

選択された libm ライブラリルーチンを最適化のためにインライン展開します。

注 – このオプションは C++ インライン関数には影響しません。

一部の libm ライブラリルーチンにはインラインテンプレートがあります。このオプションを指定すると、これらのテンプレートが選択され、現在選択されている浮動小数点オプションとプラットフォームに対してもっとも高速な実行可能コードが生成されます。

相互の関連性

-xlibmopt オプションの機能は -fast オプションを指定した場合にも含まれます。

関連項目

-fast、『数値計算ガイド』

-xlibmopt

最適化された数学ルーチンのライブラリを使用します。

パフォーマンスが最適化された数学ルーチンのライブラリを使用し、より高速で実行できるコードを生成します。通常の数学ライブラリを使用した場合とは、結果が少し異なることがあります。このような場合、異なる部分は通常は最後のビットです。

このライブラリオプションをコマンド行に指定する順序は重要ではありません。

相互の関連性

-xlibmopt オプションの機能は -fast オプションを指定した場合にも含まれます。

関連項目

-fast、-xnolibmopt

-xlic_lib=sunperf

SPARC:Sun Performance Library™ とリンクします。

-l と同様、このオプションは、ソースまたはオブジェクトファイル名に続けて、コマンド行の最後に指定する必要があります。

注 – ibrary=sunperf オプションは、ライブラリを正しい順序で確実にリンクするので Sun Performance Library のリンクにお勧めです。また、-library=sunperf オプションは位置に依存しない (コマンド行のどこにでも表示できる) ので、-staticlib を使用して Sun Performance Library を静的にリンクすることができます。-staticlib オプションは、-Bstatic -xlic_lib=sunperf -Bdynamic の組み合わせよりも便利です。

相互の関連性

-xlang オプションと -xlic_lib オプションを同じコンパイラコマンドで使用することはできません。-xlang を使用していて、しかも Sun Performance Library でリンクする必要がある場合は、代わりに -library=sunperf を使用してください。

-library=sunperf と -xlic_lib=sunperf を同じコンパイラコマンドで使用することはできません。

Sun Performance Library を静的にリンクするには、次の例にあるように、-library=sunperf と -staticlib=sunperf の各オプションを使用することをお勧めします。

```
example% CC -library=sunperf -staticlib=sunperf ... (recommended)
```

-library=sunperf の代わりに -xlic_lib=sunperf オプションを使用する場合は、次の例で示すように -Bstatic オプションを使用します。

```
% CC ... -Bstatic -xlic_lib=sunperf -Bdynamic ...
```

関連項目

_library、README ファイル『performance_library』

-xlicinfo

ライセンスサーバー情報を表示します。

このオプションは、ライセンスサーバー名と、検査済みのライセンスを所持するユーザーのユーザー ID を返します。

-xlinkopt[=*level*]

オブジェクトファイル内のあらゆる最適化のほかに、結果として出力される実行可能ファイルや動的ライブラリに対してリンク時の最適化も行うようコンパイラに指示します。このような最適化は、リンク時にオブジェクトのバイナリコードを解析することによって実行されます。オブジェクトファイルは書き換えられませんが、最適化された実行可能コードは元のオブジェクトコードとは異なる場合があります。

-xlinkopt をリンク時に有効にするには、少なくともコンパイルコマンドで -xlinkopt を使用する必要があります。-xlinkopt を指定しないでコンパイルされたオブジェクトバイナリについても、オプティマイザは限定的な最適化を実行できます。

-xlinkopt は、コンパイラのコマンド行にある静的ライブラリのコードは最適化しますが、コマンド行にある共有 (動的) ライブラリのコードは最適化しません。共有ライブラリを構築する場合 (-G でコンパイルする場合) にも、-xlinkopt を使用できます。

値

レベルには、実行される最適化レベルを 0、1、2 のいずれかで設定します。最適化レベルは次のとおりです。

表 A-37 -xlinkopt の値

リンクオプティマイザの設定	動作
0	リンクオプティマイザは無効です (デフォルト)。
1	リンク時の命令キャッシュカラーリングと分岐の最適化を含む、制御フロー解析に基づき最適化を実行します。
2	リンク時のデッドコードの除去とアドレス演算の簡素化を含む、追加のデータフロー解析を実行します。

コンパイル手順とリンク手順を別々にコンパイルする場合は、両方の手順に -xlinkopt を指定する必要があります。

```
example% cc -c -xlinkopt a.c b.c

example% cc -o myprog -xlinkopt=2 a.o
```

レベルパラメータは、コンパイラのリンク時にだけ使用されます。上記の例では、オブジェクトバイナリが指定された 1 のレベルでコンパイルされていても、リンクオプティマイザレベルは 2 です。

デフォルト

レベルパラメータなしで -xlinkopt を使用することは、-xlinkopt=1 を指定することと同じです。

相互の関連性

このオプションは、プログラム全体をコンパイルし、プロファイルフィードバックとともにプログラム全体のコンパイルを行う際に使用するのがもっとも効果的です。プロファイリングによって、コードでもっともよく使用される部分ともっとも使用されない部分が明らかにし、その結果に従ってオプティマイザを集中的に動作させること

ができます。これは、リンク時にコードの最適な配置を行うことによって、命令のキャッシュミスを低減できるような、大きなアプリケーションにとって特に重要です。このようなコンパイルの例を次に示します。

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

プロファイルフィードバックの使用方法の詳細については、397 ページの「-xprofile=p」を参照してください。

警告

リンクオプティマイザは、インクリメンタルリンカー `ild` とともに使うことはできません。`-xlinkopt` ではデフォルトのリンカーを `ld` に設定します。`-xildon` を使用してインクリメンタルリンカーを明示的に有効にし、さらに `-xlinkopt` を指定すると、`-xlinkopt` は無効になります。

`-xlinkopt` を指定してコンパイルする場合は、`-zcompreloc` リンカーオプションは使用しないでください。

このオプションを指定してコンパイルすると、リンク時間が若干増えます。オブジェクトファイルも大きくなりますが、実行可能ファイルのサイズは変わりません。`-xlinkopt` と `-g` を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

-xM

指定した C プログラムに対してプリプロセッサだけを実行します。その際、makefile 用の依存関係を生成してその結果を標準出力に出力します (`make` ファイルと依存関係についての詳細は `make`(1) のマニュアルページを参照してください)。

例

例を次に示します。

```
#include <unistd.h>
void main(void)
{}
```

この例で出力されるものは、次のとおりです。

```
e.o:e.c
e.o:/usr/include/unistd.h
e.o:/usr/include/sys/types.h
e.o:/usr/include/sys/machtypes.h
e.o:/usr/include/sys/select.h
e.o:/usr/include/sys/time.h
e.o:/usr/include/sys/types.h
e.o:/usr/include/sys/time.h
e.o:/usr/include/sys/unistd.h
```

関連項目

makefile および依存関係についての詳細は、make(1) のマニュアルページを参照してください。

-xM1

このオプションは、/usr/include ヘッダーファイルの依存関係とコンパイラで提供されるヘッダーファイルの依存関係を報告しないという点を除くと、-xM と同じです。

-xMerge

SPARC:データセグメントとテキストセグメントをマージします。

オブジェクトファイルのデータは読み取り専用です。また、このデータは ld -N を指定してリンクしない限りプロセス間で共有されます。

関連項目

ld(1) のマニュアルページ

-xmemalign=*ab*

データの境界整列についてコンパイラが使用する想定を制御するには、-xmemalign オプションを使用します。境界整列が潜在的に正しくないメモリーアクセスにつながる生成コードを制御し、境界整列が正しくないアクセスが発生したときのプログラム動作を制御すれば、より簡単に SPARC にコードを移植できます。

想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。*a* (境界整列) と *b* (動作) の両方の値が必要です。*a* は、想定する最大メモリー境界整列です。*b* は、境界整列に失敗したメモリーへのアクセスに対する動作です。

コンパイル時に境界整列が判別できるメモリーへのアクセスの場合、コンパイラはそのデータの境界整列に適したロードおよびストア命令を生成します。

コンパイル時に境界整列が判別できないメモリーへのアクセスの場合、コンパイラは必要なロードおよびストア命令を生成するための境界整列を想定する必要があります。

値

次に、-memalign の境界整列と動作の値を示します。

表 A-38 -xmemalign の境界整列と動作の値

a		*b*	
1	最大 1 バイトの境界整列	i	アクセスを解釈し、実行を継続する
2	最大 2 バイトの境界整列	s	シグナル SIGBUS を発生させる
4	最大 4 バイトの境界整列	f	4 バイト以下の境界整列に対してシグナル SIGBUS を発生させ、それ以外ではアクセスを解釈して実行を継続する
8	最大 8 バイトの境界整列		
16	最大 16 バイトの境界整列		

デフォルト

-xmemalign のデフォルトの値を次に示します。次のデフォルトの値は、-xmemalign フラグがまったく指定されていない場合にのみ適用されます。

- -xmemalign=4s: -xarch の値が generic、v7、v8、v8a、v8plus、v8plusa のいずれかの場合に適用される。
- xmemalign=8s: -xarch の値が v9 と v9a のどちらかの場合に適用される。

次に、-xmemalign フラグが指定されているが値を持たない場合のデフォルト値を示します。

- -xmemalign=1i: すべての -xarch 値に適用される。

例

次の表は、-xmemalign で処理できるさまざまな境界整列の状況とそれに適した -xmemalign 指定を示しています。

表 A-39 -xmemalign の例

コマンド	状況
-xmemalign=1s	境界整列されていないデータへのアクセスが多いため、トラップ処理が遅すぎる
-xmemalign=8i	コード内に境界整列されていないデータへのアクセスが意図的にいくつか含まれているが、それ以外は正しい
-xmemalign=8s	プログラム内に境界整列されていないデータへのアクセスは存在しないと思われる
-xmemalin=2s	奇数バイトへのアクセスが存在しないか検査したい
-xmemalign=2i	奇数バイトへのアクセスが存在しないか検査し、プログラムを実行したい

-xnativeconnect[=*i*]

-xnativeconnect オプションを使用して、オブジェクトファイル内のインタフェース情報を後続の共有ライブラリに埋め込んで、共有ライブラリを Java™ プログラミング言語で記述したコード (Java コード) から使用可能にすることができます。また、共有ライブラリを構築するときに、-G を指定して -xnativeconnect を含める必要があります。

-xnativeconnect を指定した場合は、ネイティブコードのインタフェースの外部に対する可視性が最大になります。ネイティブコネクタツール (NCT) を使用して、Java コードおよび Java Native Interface (JNI) コードを自動的に生成することができます。

NCT で -xnativeconnect を使用すると、C++ 共有ライブラリ内の関数を Java コードから呼び出すことができます。NCT の使用方法の詳細については、オンラインヘルプを参照してください。

値

i には、以下のいずれかを指定します。

表 A-40 -xnativeconnect の値

値	意味
%all	-xnativeconnet のオプションごとに異なるデータを生成します。
%none	-xnativeconnet のオプションごとに異なるデータを生成しません。
[no%]inlines	参照先のインライン関数のアウトオブラインインスタンスを生成します。これにより、ネイティブコネクタを使用して、外部から見える方法でインライン関数を呼び出すことができます。呼び出し側でのこれらの関数の通常のインライン化には影響しません。
[no%]interfaces	バイナリインタフェース記述子 (BIDS) を生成します。

デフォルト

- -xnativeconnect を指定しない場合は、コンパイラでは -xnativeconnect=%none が指定されます。
- -xnativeconnect だけを指定した場合は、コンパイラでは -xnativeconnect=inlines,interfaces が指定されます。
- このオプションは累積されません。コンパイラでは、最後に設定したものだけが有効になります。次に例を示します。

```
CC -xnativeconnect=inlines first.o -xnativeconnect=interfaces
second.o -O -G -o library.so
```

この例の場合は、コンパイラでは -xnativeconnect=no%inlines,interfaces が指定されます。

警告

-xnativeconnect を使用する場合は、-compat=4 を指定してコンパイルしないでください。引数なしで -compat を使用した場合は、コンパイラでは -compat=4 が指定されます。-compat を使用しない場合は、コンパイラでは -compat=5 が指定されます。-compat=5 を指定することで、互換性モードを明示的に設定することもできます。

-xnolib

デフォルトのシステムライブラリとのリンクを無効にします。

通常 (このオプションを指定しない場合)、C++ コンパイラは、C++ プログラムをサポートするためにいくつかのシステムライブラリとリンクします。このオプションを指定すると、デフォルトのシステムサポートライブラリとリンクするための-l*lib* オプションが ld に渡されません。

通常、コンパイラは、システムサポートライブラリにこの順序でリンクします。

- 標準モード (デフォルトモード)

```
-lCstd -lCrun -lm -lw -lcx -lc
```

- 互換モード (-compat)

```
-lC -lm -lw -lcx -lc
```

-l オプションの順序は重要です。-lm、-lw、-lcx オプションは -lc より前になければなりません。

注 – -mt コンパイラオプションを指定した場合、コンパイラは通常 -lm でリンクする直前に -lthread でリンクします。

デフォルトでどのシステムサポートライブラリがリンクされるかを知りたい場合は、コンパイルで -dryrun オプションを指定します。たとえば、次のコマンドを実行するとします。

```
example% CC foo.cc -xarch=v9 -dryrun
```

上記の出力には次の行が含まれます。

```
-lCstd -lCrun -lm -lw -lc
```

-xarch=v9 を指定したときは、-lcx がリンクされないことに注意してください。

例

C アプリケーションのバイナリインタフェースを満たす最小限のコンパイルを行う場合、つまり、C サポートだけが必要な C++ プログラムの場合は、次のように指定します。

```
example% CC -xnolib test.cc -lc
```

一般的なアーキテクチャ命令を持つシングルスレッドアプリケーションに libm を静的にリンクするには、次のように指定します。

- 標準モードの場合

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm \
-Bdynamic -lw -lcx -lc
```

- 互換モードの場合

```
example% CC -compat -xnolib test.cc -lC -Bstatic -lm \
-Bdynamic -lw -lcx -lc
```

相互の関連性

-xarch=v9、-xarch=v9a、-xarch=v9b のいずれかでリンクする場合には、使用できない静的システムライブラリがあります (libm.a や libc.a など)。

-xnolib を指定する場合は、必要なすべてのシステムサポートライブラリを手動で一定の順序にリンクする必要があります。システムサポートライブラリは最後にリンクしなければなりません。

-xnolib を指定すると、-library は無視されます。

警告

C++ 言語の多くの機能では、libC (互換モード) または libCrun (標準モード) を使用する必要があります。

このリリースのシステムサポートライブラリは安定していないため、リリースごとに変更される可能性があります。

-lcx は 64 ビットコンパイルモードにはありません。

関連項目

-library、 -staticlib、-l

-xnolibmil

コマンド行の -xlibmil を取り消します。

最適化された数学ライブラリとのリンクを変更するには、このオプションを -fast と一緒に使用してください。

-xnolibmopt

数学ルーチンのライブラリを使用しないようにします。

例

次の例のように、このオプションはコマンド行で -fast オプションを指定した場合は、その後に使用してください。

```
example% CC -fast -xnolibmopt
```

-xO*level*

最適化レベルを指定します。大文字 O の後に数字の 1、2、3、4、5 のいずれかが続きます。一般的に、プログラムの実行速度は最適化のレベルに依存します。最適化レベルが高いほど、実行時のパフォーマンスは向上します。しかし、最適化レベルが高ければ、それだけコンパイル時間が増え、実行可能ファイルが大きくなる可能性があります。

ごくまれに、-xO2 の方が他の値より実行速度が速くなることがあり、-xO3 の方が -xO4 より速くなることがあります。すべてのレベルでコンパイルを行なってみて、こうしたことが発生するかどうか試してみてください。

メモリー不足になった場合、オプティマイザは最適化レベルを落として現在の手続きをやり直すことによってメモリー不足を回復しようとします。ただし、以降の手続きについては、-xO*level* オプションで指定された最適化レベルを使用します。

-xO には 5 つのレベルがあります。以降では各レベルが SPARC および x86 プラットフォームでどのように動作するかを説明します。

値

SPARC プラットフォームの場合

- -xO1 では、最小限の最適化 (ピープホール) が行われます。これはコンパイルの後処理におけるアセンブリレベルでの最適化です。-xO2 や -xO3 を使用するとコンパイル時間が著しく増加する場合や、スワップ領域が不足する場合だけ -xO1 を使用してください。
- -xO2 では、次の基本的な局所的および大域的な最適化が行われます。
  - 帰納的変数の削除
  - 局所的および大域的な共通部分式の削除
  - 計算の簡略化

- コピーの伝播
- 定数の伝播
- ループ不変式の最適化
- レジスタ割り当て
- 基本ブロックのマージ
- 末尾再帰の削除
- デッドコードの削除
- 末尾呼び出しの削除
- 複雑な式の展開

このレベルでは、外部変数や間接変数の参照や定義は最適化されません。

`-O` は `-xO2` を指定することと同じです。

- `-xO3` では、`-xO2` レベルで行う最適化に加えて、外部変数に対する参照と定義も最適化されます。このレベルでは、ポインタ代入の影響は追跡されません。`volatile` で適切に保護されていないデバイスドライバをコンパイルする場合か、シグナルハンドラの中から外部変数を修正するプログラムをコンパイルする場合は、`-xO2` を使用してください。一般に、このレベルを使用するとコードサイズが大きくなります。スワップ領域が不足する場合は、`-xO2` を使用してください。

- `-xO4` では、`-xO3` レベルで行う最適化レベルに加えて、同じファイルに含まれる関数のインライン展開も自動的に行われます。インライン展開を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。一般に、このレベルを使用するとコードサイズが大きくなります。スワップ領域が不足する場合は、`-xO2` を使用してください。

- `-xO5` では、最高レベルの最適化が行われます。これを使用するのは、コンピュータのもっとも多くの時間を小さなプログラムが使用している場合だけにしてください。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。このレベルの最適化によってパフォーマンスが改善される確率を高くするには、プロファイルのフィードバックを使用します。397 ページの「`-xprofile=p`」を参照してください。

x84 プラットフォームの場合:

- `-xO1` では、基本的な最適化を行います。このレベルには、計算の簡略化、レジスタ割り当て、基本ブロックのマージ、デッドコードとストアの削除、およびピープホールの最適化が含まれます。

- -xO2 では、局所的な共通部分の削除、局所的なコピーと定数の伝播、末尾再帰の削除、およびレベル 1 で行われる最適化を実行します。
- -xO3 では、局所的な共通部分の削除、大域的なコピーと定数の伝播、ループ強度低下、帰納的変数の削除、およびループ不変式の最適化、およびレベル 2 で行われる最適化を実行します。
- -xO4 では、レベル 3 で行う最適化に加えて、同じファイルに含まれる関数のインライン展開も自動的に行われます。インライン展開を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。このレベルでは一般用のフレームポインタ登録 (edp) も解放します。一般にこのレベルを使用するとコードサイズが大きくなります。
- -xO5 では、最高レベルの最適化が行われます。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。

相互の関連性

-g または -g0 を使用するとき、最適化レベルが -xO3 以下の場合、最大限のシンボリック情報とほぼ最高の最適化が得られます。末尾呼び出しの最適化とバックエンドのインライン化は無効です。

-g または -g0 を使用するとき、最適化レベルが -xO4 以上の場合、最大限のシンボリック情報と最高の最適化が得られます。

-g を指定してデバッグを行なっても -xO*level* には影響はありません。しかし、-xO*level* によって -g がある程度の制限を受けます。たとえば、-xO*level* オプションを使用すると、dbx から渡された変数を表示できないなど、デバッグの機能が一部制限されます。しかし、dbx where コマンドを使用して、シンボリックトレースバックを表示することは可能です。詳細は、『dbx コマンドによるデバッグ』を参照してください。

-xcrossfile オプションは、-xO4 または -xO5 と一緒に使用した場合にのみ効果があります。

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 では、-xinline オプションを指定したかどうかは関係なく、オプティマイザはどの関数をインライン化するかを判断します。-xO4 では、コンパイラはどの関数が、イン

ライン化されたときにパフォーマンスを改善するかを判断しようとします。
-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

デフォルト

デフォルトでは最適化は行われません。ただし、これは最適化レベルを指定しない場合に限り有効です。最適化レベルを指定すると、最適化を無効にするオプションはありません。

最適化レベルを設定しないようにする場合は、最適化レベルを示すようなオプションを指定しないようにしてください。たとえば、-fast は最適化を -xO5 に設定するマクロオプションです。それ以外で最適化レベルを指定するすべてのオプションでは、最適化レベルが設定されたという警告メッセージが表示されます。最適化を設定せずにコンパイルする唯一の方法は、コマンド行または make ファイルから最適化レベルを指定するオプションをすべて削除することです。

警告

大規模な手続き (数千行のコードからなる手続き) に対して -xO3 または -xO4 を指定して最適化をすると、途方もない大きさのメモリーが必要になり、マシンのパフォーマンスが低下することがあります。

こうしたパフォーマンスの低下を防ぐには、limit コマンドを使用して、1 つのプロセスで使用できる仮想メモリーの大きさを制限します (csh(1) のマニュアルページを参照)。たとえば、使用できる仮想メモリーを 16M バイトに制限するには、次のコマンドを使用します。

```
example% limit datasize 16M
```

このコマンドにより、データ領域が 16M バイトに達したときに、オプティマイザがメモリー不足を回復しようとします。

マシンが使用できるスワップ領域の合計容量を超える値は、制限値として指定することはできません。制限値は、大規模なコンパイル中でもマシンの通常の使用ができるぐらいの大きさにしてください。

最良のデータサイズ設定値は、要求する最適化のレベルと実メモリーの量、仮想メモリーの量によって異なります。

現在のスワップ領域を表示するには次のように入力します。**swap -l**

現在の実メモリーを表示するには次のように入力します。**dmesg | grep mem**

関連項目

-xldscope -fast、-xcrossfile=*n*、-xprofile=*p*、csh(1) のマニュアルページ

-xopenmp[=*i*]

SPARC:OpenMP 指令による明示的並列化を使用するには、-xopenmp オプションを指定します。この実装には、ソースコード指令、実行時ライブラリルーチン、および環境変数が含まれています。

値

次に、*i*: の値を示します。

表 A-41 -xopenmp の値

i の値	意味
parallel	OpenMP プラグマの認識を有効にします。-xopenmp=parallel での最低最適化レベルは -x03 です。コンパイラは必要に応じて低い最適化レベルを -x03 に引き上げ、警告メッセージを表示します。
stubs	OpenMP プラグマの認識を無効にし、スタブライブラリルーチンとリンクしますが、最適化レベルは変更しません。アプリケーションが OpenMP 実行時ライブラリルーチンを明示的に呼び出していて、逐次実行するようコンパイルしたい場合にこのオプションを使用してください。-xopenmp=stubs コマンドは _OPENMP プリプロセッサトークンも定義します。
none	OpenMP プラグマを認識せず、プログラムの最適化レベルを変更せず、プリプロセッサトークンを事前定義しません。

デフォルト

-xopenmp を指定しない場合は、コンパイラでは -xopenmp=none が指定されます。

-xopenmp は指定されているけれども引数は指定されていない場合、コンパイラはこのオプションを -xopenmp=parallel と設定します。

警告

-xopenmp のデフォルトは、将来変更される可能性があります。警告メッセージを出力しないようにするには、適切な最適化を明示的に指定します。

コンパイルとリンクを別々に実行する場合、リンク手順にも -xopenmp を指定してください。これは、OpenMP 指令を含むライブラリをコンパイルする場合に特に重要です。

関連項目

多重処理アプリケーションを構築するために使用する OpenMP Fortran 95、C、C++ アプリケーションプログラミングインターフェース (API) の概要については、『OpenMP API ユーザーズガイド』を参照してください。

-xpagesize=*n*

*(SPARC)*スタックとヒープ用に優先ページサイズを設定します。

値

n には次の値のいずれかを指定します。8K、64K、512K、4M、32M、256M、2G、16G、default のいずれか。

ターゲットプラットフォームには、getpagesize(3C) によって返される、Solaris オペレーティング環境における有効なページサイズを指定する必要があります。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。Solaris オペレーティング環境では、ページサイズ要求に従うという保証はありません。

pmap(1) または meminfo(2) を使用すれば、ターゲットプラットフォームのページサイズを確認できます。

注 – この機能は Solaris 7 および Solaris 8 オペレーティング環境では使用できません。このオプションを指定してコンパイルされたプログラムは、Solaris 7 および Solaris 8 オペレーティング環境ではリンクしません。

デフォルト

-xpagesize=default を指定すると、Solaris オペレーティング環境はページサイズを設定します。引数を指定せずに -xpagesize を指定すると、-xpagesize=default を指定したことになります。

拡張

このオプションは -xpagesize_heap と -xpagesize_stack のマクロです。これらの 2 つのオプションは -xpagesize と同じ次の引数を使用します。8K、64K、512K、4M、32M、256M、2G、16G、default のいずれか。両方に同じ値を設定するには -xpagesize を指定します。別々の値を指定するには個々に指定します。

警告

-xpagesize オプションは、コンパイル時とリンク時に使用しない限り無効です。

関連項目

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris 9 コマンドの ppgsz(1) を実行するのと同じことです。詳細については、Solaris 9 のマニュアルページを参照してください。

-xpagesize_heap=*n*

*(SPARC)*ヒープの希望ページサイズを設定します。

値

n の値は、8K、64K、512K、4M、32M、256M、2G、16G、または default です。ターゲットプラットフォームには、getpagesize(3C) が返す、Solaris オペレーティング環境における有効なページサイズを指定する必要があります。有効なページサイズを指定しなかった場合、要求は、メッセージが出力されないまま実行時に無視されます。

ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

注 – このオプションを指定してコンパイルされたプログラムは、Solaris 7 および Solaris 8 オペレーティング環境ではリンクしません。

デフォルト

`-xpagesize_heap=default` を指定すると、Solaris オペレーティング環境がページサイズを設定します。引数を指定しないで `-xpagesize_heap` を指定すると、`-xpagesize_heap=default` を指定したことになります。

関連項目

このオプションを指定してコンパイルするのは、`LD_PRELOAD` 環境変数を同等のオプションで `mpss.so.1` に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris 9 コマンドの `ppgsz`(1) を実行するのと同じことです。詳細については、Solaris 9 のマニュアルページを参照してください。

-xpagesize_stack=*n*

*(SPARC)*スタックの希望ページサイズを設定します。

値

n の値は、`8K`、`64K`、`512K`、`4M`、`32M`、`256M`、`2G`、`16G`、または `default` です。ターゲットプラットフォームには、`getpagesize`(3C) が返す、Solaris オペレーティング環境における有効なページサイズを指定する必要があります。有効なページサイズを指定しなかった場合、要求は、メッセージが出力されないまま実行時に無視されます。

ターゲットプラットフォームのページサイズを判断するには、`pmap`(1) または `meminfo`(2) を使用します。

注 – このオプションを指定してコンパイルされたプログラムは、Solaris 7 および Solaris 8 オペレーティング環境ではリンクしません。

デフォルト

-xpagesize_stack=default を指定すると、Solaris オペレーティング環境がページサイズを設定します。引数を指定しないで -xpagesize_stack を指定すると、-xpagesize_stack=default を指定したことになります。

関連項目

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris 9 コマンドの ppgsz(1) を実行するのと同じことです。詳細については、Solaris 9 のマニュアルページを参照してください。

-xpch=*v*

このコンパイラオプションは、プリコンパイル済みヘッダー機能を有効にします。プリコンパイル済みヘッダー機能は、ソースファイルが、大量のソースコードを含む一連の共通インクルードファイル群を共有しているようなアプリケーションのコンパイル時間を短縮させることができます。コンパイラは 1 つのソースファイルから一連のヘッダーファイルに関する情報を収集し、そのソースファイルを再コンパイルしたり、同じ一連のヘッダーファイルを持つ他のソースファイルをコンパイルしたりするときにその情報を使用します。コンパイラが収集する情報は、プリコンパイル済みヘッダーファイルに格納されます。この機能を利用するには、-xpch と -xpchstop のオプションを #pragma hdrstop 指令とともに使用します。

関連項目

- 387 ページの「-xpchstop=file」
- C ユーザーズガイドの “hdrstop”

プリコンパイル済みヘッダーファイルの作成

-xpch=*v* を指定する場合、*v* には collect:*pch_filename* または use:*pch_filename* を指定します。-xpch を初回に使用するときは、collect モードを指定する必要があります。-xpch=collect を指定するコンパイルコマンドは、ソースファイルを 1 つしか指定できません。次の例では、-xpch オプションがソースファイル a.cc に基づいて myheader.Cpch というプリコンパイル済みヘッダーファイルを作成します。

```
CC -xpch=collect:myheader a.cc
```

有効なプリコンパイル済みヘッダーファイル名には必ず、.cpch という接尾辞が付きます。*pch_filename* を指定する場合、自分で接尾辞を追加することも、コンパイラに追加させることもできます。たとえば、`cc -xpch=collect:foo a.cc` と指定すると、プリコンパイル済みヘッダーファイルには `foo.Cpch` という名前が付けられます。

プリコンパイル済みヘッダーファイルを作成する場合、プリコンパイル済みヘッダーファイルを使用するすべてのソースファイルで共通な、一連のインクルードファイルを含むソースファイルを選択します。インクルードファイルの並びは、これらのソースファイル全体で同一でなければなりません。`collect` モードでは、1 つのソースファイル名だけが有効な値である点に注意してください。たとえば、`CC -xpch=collect:foo bar.cc` は有効ですが、`CC -xpch=collect:foo bar.cc foobar.cc` は、2 つのソースファイルを指定しているので無効です。

プリコンパイル済みヘッダーファイルの使用方法

プリコンパイル済みヘッダーファイルを使用するには、`-xpch=use:`*pch_filename* と指定します。プリコンパイル済みヘッダーファイルを作成するために使用されたソースファイルと同じインクルードファイルの並びを持つソースファイルであれば、いくつでも指定できます。たとえば、`use` モードで、次のようなコマンドがあるとします。 `CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc`.

以下の項目で真ではないものがあれば、プリコンパイル済みヘッダーファイルを再作成する必要があります。

- プリコンパイル済みヘッダーファイルにアクセスするために使用するコンパイラは、プリコンパイル済みヘッダーファイルを作成したコンパイラと同じであること。あるバージョンのコンパイラで作成されたプリコンパイル済みヘッダーファイルは、インストールされているパッチが起因する違いなどから、別のバージョンのコンパイラでは使用できない場合があります。
- `-xpch` オプション以外で `-xpch=use` とともに指定するコンパイラオプションは、プリコンパイル済みヘッダーファイルが作成されたときに指定されたオプションと一致すること。
- `-xpch=use` を指定する時にインクルードされるヘッダー群は、プリコンパイル済みヘッダーファイルが作成されたときに指定されたヘッダー群と同じであること。

- -xpch=use で指定するインクルードされるヘッダーの内容が、プリコンパイル済みヘッダーファイルが作成されたときに指定されたインクルードされるヘッダーの内容と同じであること。
- 現在のディレクトリ (すなわち、コンパイルが実行中で指定されたプリコンパイル済みヘッダーファイルを使用しようとしているディレクトリ) が、プリコンパイル済みヘッダーファイルが作成されたディレクトリと同じであること。
- -xpch=collect で指定したファイル内の #include 指令を含む前処理指令の最初のシーケンスが、-xpch=use で指定するファイル内の前処理指令のシーケンスと同じであること。

プリコンパイル済みヘッダーファイルを複数のソースファイル間で共有するために、これらのソースファイルには、最初のトークンの並びとして一連の同じインクルードファイルを使用していなければなりません。この最初のトークンの並びは、活性文字列 (viable prefix) として知られています。活性文字列は、同じプリコンパイル済みヘッダーファイルを使用するすべてのソースファイル間で一貫して解釈される必要があります。

活性文字列は各ソースファイルの最初のトークンで始まり、#pragma hdrstop または -xpchstop オプションで指定されるヘッダーファイルの #include の最後のトークンで終わります。

ソースファイルの活性文字列には、コメントと次に示すプリプロセッサ指令のみを指定できます。

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

上記の任意の指令はマクロを参照する場合があります。#else、#elif、#endif 指令は、活性文字列の中で一致している必要があります。

プリコンパイル済みヘッダーファイルを共有する各ファイルの活性文字列の中では、対応する各 #define 指令と #undef 指令は同じシンボルを参照する必要があります (#define の場合は、各指令は同じ値を参照しなければなりません)。各活性文字列の

中での順序も同じである必要があります。対応する各プラグマも同じで、その順序もプリコンパイル済みヘッダーを共有するすべてのファイルで同じでなければなりません。

プリコンパイル済みヘッダーファイルに組み込まれるヘッダーファイルは、次の項目に違反しないようにしてください。これらの制限に違反するプログラムをコンパイルすると、結果は未定義です。

- ヘッダーファイルには、関数や変数の定義を含めることはできません。
- ヘッダーファイルは、__DATE__ と __TIME__ は使用できません。これらのプリプロセッサマクロを使用すると、予測できない結果になります。
- ヘッダーファイルには、#pragma hdrstop を含めることはできません。
- ヘッダーファイルは、活性文字列内で __LINE__ と __FILE__ を使用できません。インクルードヘッダー内の __LINE__ と __FILE__ は使用できます。

make ファイルの変更方法

-xpch を構築に組み込むために make ファイルを変更する方法としては、以下があります。

- make と dmake の KEEP_STATE 機能と CCFLAGS 補助変数を使用すれば、暗黙的な make 規則を使用できます。プリコンパイル済みヘッダーが、独立したステップとして出力されます。

```
.KEEP_STATE :
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch : foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out : foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

また、CCFLAGS 補助変数を使用する代わりに、独自のコンパイル規則を定義することもできます。

```
.KEEP_STATE :
.SUFFIXES : .o .cc
%.o : %.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch : foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out : foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- KEEP_STATE を使用せずに、通常のコンパイルの二次効果としてプリコンパイル済みヘッダーを生成できますが、それには、明示的なコンパイルコマンドを使用する必要があります。

```
shared.Cpch + foo.o : foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o : ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o : pong.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out : foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

-xpchstop=*file*

-xpchstop=*file* オプションは、-xpch オプションを指定してプリコンパイル済みヘッダーファイルを作成する際の最後のインクルードファイルを指定します。コマンド行で -xpchstop を使用することは、cc コマンドで指定する各ソースファイル内のファイルを参照する最初のインクルード指令の後に、hdrstop プラグマを配置することと同じです。

次の例では、-xpchstop オプションで、プリコンパイル済みヘッダーファイルの実行可能な接頭辞が projectheader.h をインクルードして終わるよう指定してます。したがって、privateheader.h は活性文字列の一部ではありません。

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h
-c
```

関連項目

-xpch, pragma hdrstop

-xpg

-xpg オプションでは、gprof で自動プロファイル処理するためのデータを収集するコードが生成されます。このオプションを指定すると、プログラムが正常に終了したときに gmon.out を生成する実行時記録メカニズムが呼び出されます。

警告

コンパイルとリンクを別々に行う場合は、-xpg でコンパイルしたときは -xpg でリンクする必要があります。

関連項目

-xprofile=*p*、analyzer(1) のマニュアルページ、『プログラムのパフォーマンス解析』

-xport64 [=(*v*)]

このオプションを指定すると、64 ビット環境に移植するコードをデバッグできます。このオプションは、具体的には、V7 などの 32 ビットアーキテクチャを V9 などの 64 ビットアーキテクチャにコード移植する際によく見られる、型 (ポインタを含む) の切り捨て、符号の拡張、ビット配置の変更といった問題について警告します。

値

次に、*v*: に指定できる値を示します。

表 A-42 `-xport64` の値

v の値	意味
no	32 ビット環境から 64 ビット環境へのコード移植に関し、まったく警告を生成しない。
implicit	暗黙の変換に関してのみ警告を生成する。明示的なキャストが存在する場合には警告を生成しない。
full	32 ビット環境から 64 ビット環境へのコード移植に関し、あらゆる警告を生成する。具体的には、64 ビット値の切り捨て、ISO 値保護規則に基づく 64 ビットへの符号拡張、ビットフィールドの配置の変更などです。

デフォルト

`-xport64` を指定しない場合のデフォルトは、`-xport64=no` です。`-xport64` を値なしで指定した場合は、コンパイラでは `-xport64=full` が指定されます。

例

以降では、型の切り捨て、符号の拡張、ビット配置の変更を行うコード例を紹介します。

64 ビット値の切り捨てのチェック

V9 などの 64 ビットアーキテクチャを移植する場合、データが切り捨てられることがあります。切り捨ては、初期化時に代入によって暗黙的に行われることもあれば、明示的なキャストによって行われることもあります。2 つのポインタの違いは typedef

ptrdiff_t であり、32 ビットモードでは 32 ビット整数型、64 ビットモードでは 64 ビット整数型です。大きいサイズから小さいサイズの整数型に切り捨てると、次の例にあるような警告が生成されます。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to
"int" causes truncation.
1 Warning(s) detected.
example%
```

明示的キャストによってデータが切り捨てられている場合に、64 ビットのパイルモードでの切り捨て警告を抑止するには、-xport64=implicit を使用します。

```
example% CC -c -xarch=v9 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to
"int" causes truncation.
1 Warning(s) detected.
example%
```

64 ビットアーキテクチャへの移植でよく発生するもう 1 つの問題として、ポインタの切り捨てがあります。これは常に、C++ におけるエラーです。切り捨てを引き起こす、ポインタのキャスティングなどの操作は、-xport64 を指定した場合に V9 ではエラー診断となります。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) ( ((unsigned int)p) & 0xFF ); // -xarch=v9 error
  return 0;
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3: Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

符号拡張のチェック

符号なし整数型の式において、通常の ISO C 値保護規則が符号付き整数値の符号拡張に対処している状況があるかどうかを、-xport64 オプションを使用してチェックすることもできます。こういった符号拡張は、実行時に微妙なバグの原因となる可能性があります。

```
example% cat test3.c
int i= -1;
void promo(unsigned long l) { }

int main() {
    unsigned long l;
    l = i;  // warn
    promo(i);       // warn
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6: Warning: Sign extension from "int" to 64-bit
integer.
"test3.c", line 7: Warning: Sign extension from "int" to 64-bit
integer.
2 Warning(s) detected.
```

ビットフィールドの配置変更のチェック

長いビットフィールドに対する警告を生成するには、-xport64 を使用します。こういったビットフィールドが存在していると、ビットフィールドの配置が大きく変わることがあります。ビットフィールド配置方式に関する前提事項に依存しているプログラムを、64 ビットアーキテクチャに問題なく移植できるためには、あらかじめ確認作業を行う必要があります。

```
example% cat test4.c
#include <stdio.h>

union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx  u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

V9 における出力

```
example% u.buf[0] = fffffffff000000  u.buf[1] = 0
```

V7 における出力

```
example% u.buf[0] = fffff000  u.buf[1] = fffff000
example% CC -c -xarch=v9 -Qoption ccfe -xport64 test4.c
"test4.c", line 5: Warning: 64-bit type bitfield may change
bitfield packing within structure or union.
"test4.c", line 6: Warning: 64-bit type bitfield may change
bitfield packing within structure or union.
2 Warning(s) detected.
example%
```

警告

-arch=generic64 や -xarch=v9 などのオプションを指定して 64 ビットモードでコンパイルしたときだけに、警告が生成されることがわかります。

関連項目

326 ページの「-xarch=isa」

-xprefetch[=*a*[,*a*...]]

SPARC:先読みをサポートするアーキテクチャ (UltraSPARC II など) で先読み命令を有効にします (-xarch=v8plus、v9plusa、v9、v9a、v9b のいずれか)。

a には次のいずれかを指定します。

表 A-43 -xprefetch の値

値	意味
auto	先読み命令の自動的な生成を有効にします。
no%auto	先読み命令の自動的な生成を無効にします。
explicit	明示的な先読みマクロを有効にします。
no%explicit	明示的な先読みマクロを無効にします。
latx:*factor*	指定された factor によってコンパイラで使用されるロードするための先読みとストアするための先読みを調整します。係数には必ず正の浮動小数点または整数を指定します。
yes	-xprefetch=yes は -xprefetch=auto,explicit と同じです。
no	-xprefetch=no は -xprefetch=no%auto,no%explicit と同じです。

-xprefetch、-xprefetch=auto、および -xprefetch=yes を指定すると、コンパイラは先読み命令をコンパイラで生成するコードに自由に挿入できます。これによって、先読みをサポートするアーキテクチャーのパフォーマンスが向上する場合があります。

計算上複雑なコードを大型のマルチプロセッサで実行しているときに `-xprefetch=latx:`*factor* を使用すると役立つ場合があります。このオプションはコードジェネレータに先読みおよび指定された係数による関連のロードやストアを行う間のデフォルトの応答時間を調整します。

先読み応答時間は、先読み命令の実行と先読みしているデータがキャッシュ内で有効になっている時間との間のハードウェアによる遅延のことです。コンパイラでは、先読み命令と先読みしたデータを使用するロード命令やストア命令の間隔を決める先読み応答時間の値が使用されます。

注 – 先読み命令とロード命令との間の使用応答時間は、先読み命令とストア命令との間の使用応答時間と異なる場合があります。

コンパイラでは、広範囲なマシンやアプリケーション間で最高のパフォーマンスが得られるように先読み機構を調整します。この調整は必ずしも最高でない場合もあります。メモリーをたくさん使用するアプリケーション、特に大型のマルチプロセッサで実行されるアプリケーションの場合は、先読み応答時間の値を増やすことでパフォーマンスを向上できる場合があります。この値を増やすには、1 よりも大きい係数を使用します。.5 と 2.0 の間の値は、おそらく最高のパフォーマンスを提供します。

外部キャッシュの中に完全に常駐するデータセットを持つアプリケーションの場合は、先読み応答時間の値を減らすことでパフォーマンスを向上できる場合があります。値を減らすには、1 未満の係数を使用します。

`-xprefetch=latx:`*factor* オプションを使用するには、1.0 に近い係数の値から始め、アプリケーションに対してパフォーマンステストを実施します。その後で係数を増減して、パフォーマンスを再度実施します。こうして最高のパフォーマンスが得られるまで係数を調整しながらパフォーマンステストを継続します。係数を小刻みに増減すると、ほんの数刻み増減しただけではパフォーマンスに違いは見られませんが、突然パフォーマンスが大きく変わり、その後再度横這い状態になります。

デフォルト

`-xprefetch` を指定しないと、`-xprefetch=no%auto,explicit` が使用されます。

`-xprefetch` だけを指定すると、`-xprefetch=auto,explicit` が使用されます。

-xprefetch だけを指定した場合や引数として auto または yes を指定した場合以外は、デフォルトで no%auto が使用されます。たとえば、-xprefetch=explicit は -xprefetch=explicit,no%auto と同じことです。

no%explicit か no を指定した場合以外は、デフォルトで explicit が使用されます。たとえば、-xprefetch=auto は -xprefetch=auto,explicit と同じことです。

-prefetch または -prefetch=yes などで自動先読みを有効にしても、応答時間係数を指定しないと、-xprefetch=latx:1.0 が使用されます。

相互の関連性

このオプションは、置き換えられる代わりに蓄積されます。

sun_prefetch.h ヘッダーファイルには、明示的な先読み命令を指定するためのマクロが含まれています。先読み命令は、実行コード中のマクロの位置にほぼ相当するところに挿入されます。

明示的な先読み命令を使用するには、使用するアーキテクチャが適切なもので、sun_prefetch.h をインクルードし、かつ、コンパイラコマンドに -xprefetch が指定されていないか、-xprefetch、xprefetch=auto、explict、-xprefetch=explict あるいは -xprefetch=yes が指定されていなければなりません。

マクロが呼び出され、sun_prefetch.h ヘッダーファイルがインクルードされていても、-xprefetch=no%explict か -xprefetch=no が指定されていると、明示的な先読み命令は実行コードに組み込まれません。

latx:*factor* の使用は、自動先読みが有効になっている場合に限り有効です。つまり、latx:*factor* は、-xprefetch=yes,latx:*factor* の場合のように、yes または auto の関係で使用しない限り無視されます。

警告

明示的な先読み命令の使用は、パフォーマンスが実際に向上する特別な場合に限定してください。

コンパイラは、広範囲なマシンやアプリケーション間で最適なパフォーマンスを得るために先読み機構を調整しますが、-xprefetch=latx:*factor* は、パフォーマンステストで明らかに利点があることが確認された場合に限り使用してください。使用先読み応答時間は、リリースごとに変わる可能性があります。したがって、別のリリースに切り替えたら、その都度応答時間係数の影響を再テストすることを推奨します。

-xprefetch_level[=*i*]

新しい -xprefetch_level=*i* オプションを使用して、-xprefetch=auto で定義した先読み命令の自動挿入を調整することができます。-xprefetch_level が高くなるほど、コンパイラはより攻撃的に、つまりより多くの先読みを挿入します。

-xprefetch_level に適した値は、アプリケーションでのキャッシュミス数によって異なります。-xprefetch_level の値を高くするほど、キャッシュミスが多いアプリケーションの性能が向上する可能性が高くなります。

値

i には 1、2、3 のいずれかを指定します。

表 A-44 -xprefecth_level の値

値	意味
1	先読み命令の自動的な生成を有効にします。
2	-xprefetch_level=1 の対象以外にも、先読み挿入対象のループを追加します。-xprefetch_level=1 で挿入された先読み以外に、先読みが追加されることがあります。
3	-xprefetch_level=2 の対象以外にも、先読み挿入対象のループを追加します。-xprefetch_level=2 で挿入された先読み以外に、先読みが追加されることがあります。

デフォルト

デフォルトは、-xprefetch=auto を指定した場合は -xprefetch_level=2 になります。

相互の関連性

このオプションは、-xprefetch=auto を指定し、最適化レベルを 3 (-xO3) 以上に設定して、先読みをサポートするプラットフォーム (v8plus、v8plusa、v9、v9a、v9b、generic64、native64) でコンパイルした場合にだけ有効です。

-xprofile=*p*

このオプションを使用して実行頻度データを収集して保存することにより、そのデータを使用してパフォーマンスを向上させることができます。このオプションは、-x02 以上の最適化のレベルを指定した場合にのみ有効です。

高い最適化レベル (-xO5 など) を指定したコンパイルは、コンパイラに実行時のパフォーマンスフィードバックを提供することで拡張されます。実行時のパフォーマンスフィードバックを生成させるためには、-xprofile=collect を指定してコンパイルし、標準的なデータセットで実行可能ファイルを実行し、次に最高の最適化レベルと -xprofile=use を指定して再コンパイルします。

プロファイルの収集は、マルチスレッド対応のアプリケーションにとって安全です。すなわち、独自のマルチタスク (-mt) を実行するプログラムをプロファイリングすることで、正確な結果が得られます。このオプションは、-x02 以上の最適化のレベルを指定した場合にのみ有効です。

値

p は次のいずれかでなければなりません。

- collect[:*name*]

実行頻度のデータを集めて保存します。後に -xprofile=use を指定した場合にオプティマイザがこれを使用します。コンパイラは、コードを生成して実行頻度を計ります。

name には分析するプログラム名を指定します。*name* は省略可能で、省略すると実行可能ファイル名は a.out とみなされます。

-xprofile=collect:*name* でコンパイルしたプログラムは、実行時に、実行時のフィードバック情報を書き込むサブディレクトリ *name*.profile を作成します。データは、このサブディレクトリのファイル feedback に書き込まれます。

$SUN_PROFDATA 環境変数と $SUN_PROFDATA_DIR 環境変数を使用すると、フィードバック情報の置き場所を変更できます。詳細は、「相互の関連性」の節を参照してください。

プログラムを複数回実行する場合、実行頻度のデータがフィードバックファイルに蓄積されます。つまり、過去のプログラム実行の出力が失われることはありません。

別々の手順でコンパイルしてリンクする場合は、-xprofile=collect を指定してコンパイルしたオブジェクトファイルは、リンクでも必ず -xprofile=collect を指定してください。

- use[:*name*]

-xprofile=collect でコンパイルしたプログラムを前回実行したときに作成されたフィードバックファイルに保存された実行頻度のデータに基づいて、プログラムが最適化されます。

name には分析する実行可能ファイル名を指定します。*name* は省略可能で、省略すると実行可能ファイル名は a.out とみなされます。

-xprofile オプション (-xprofile=collect から -xprofile=use に変わります) を除き、ソースファイルおよびコンパイラの他のオプションは、フィードバックファイルを生成したコンパイル済みプログラムのコンパイルに使用したものと完全に同一のものを指定する必要があります。同じバージョンのコンパイラは、収集構築と使用構築の両方に使用する必要があります。-xprofile=collect:*name*を使用してコンパイルする場合は、最適化コンパイルでも同じ名前 (*name*) が使用されていなければなりません。-xprofile=use:*name*

オブジェクトファイルとそのプロファイルデータの関連付けは、-xprofile=collect を指定してコンパイルしたときのオブジェクトファイルの UNIX パス名に基づいています。場合によっては、コンパイラはオブジェクトファイルとそのプロファイルデータの関連付けを行いません。前回 -xprofile=collect を指定してコンパイルされなかったためオブジェクトファイルにプロファイルデータがない場合、オブジェクトファイルが -xprofile=collect でプログラムにリンクされていない場合、プログラムが一度も実行されていない場合などです。

さらに、オブジェクトファイルが前回異なるディレクトリ内で -xprofile=collect を指定してコンパイルされ、-xprofile=collect でコンパイルされた他のオブジェクトファイルと共通ベース名を共有しているが、それらのファイルを格納しているディレクトリの名前によって一意に識別できない場合も、コンパイラは正しく処理できなくなります。この場合、オブジェクトファイルにプロファイルデータがあって

も、オブジェクトファイルが -xprofile=use で再コンパイルされたときに、コンパイラはフィードバックディレクトリ内でそのプロファイルデータを見つけることができません。

このようなあらゆる状況によって、コンパイラはオブジェクトファイルとプロファイルデータの関連付けを失います。したがって、オブジェクトファイルがプロファイルデータを持っているのに、-xprofile=use を指定したときにコンパイラがプロファイルデータをオブジェクトファイルのパス名に関連付けできない場合は、-xprofile_pathmap オプションを使用して正しいディレクトリを特定します。401 ページの「-xprofile_pathmap」参照

- tcov

「新しい」形式の tcov を使った基本ブロックカバレージ解析。

tcov の基本ブロックプロファイルの新しい形式です。-xa オプションと類似した機能を持つが、ヘッダーファイルにソースコードが含まれているプログラムや、C++ テンプレートを使用するプログラムのデータを集めます。コード生成は -xa オプションと類似していますが、.d ファイルは生成されません。その代わりにファイルが 1 つ生成されます。このファイルの名前は最終的な実行可能ファイルに基づきます。たとえば、/foo/bar にある myprog を実行する場合、データファイルは /foo/bar/myprog.profile/tcovd に保存されます。

tcov を実行する場合は、新しい形式のデータが使用されるように -x オプションを指定します。-x オプションを指定しないと、デフォルトで古い形式の .d ファイルが使用され、予期しない結果が出力されます。

-xa オプションの場合とは異なり、TCOVDIR 環境変数はコンパイル時には影響力を持ちません。ただし、TCOVDIR 環境変数の値はプログラムの実行時に使用されます。

相互の関連性

-xprofile=tcov と -xa オプションは、同じ実行可能ファイル内に指定することができます。すなわち、-xprofile=tcov でコンパイルされたファイルと -xa でコンパイルされたファイルが両方含まれたプログラムをリンクすることができます。1 つのファイルを両方のオプションでコンパイルすることはできません。

-xinline か-xO4 を使用したために、関数のインライン化が行われている場合は、-xprofile=tcov によって生成されたコードカバレージ報告は信用できない可能性があります。

環境変数の $SUN_PROFDATA と $SUN_PROFDATA_DIR を設定して -xprofile=collect を指定してコンパイルされたプログラムがどこにプロファイルデータを入れるかを制御できます。これらの変数をまだ設定していない場合は、プロファイルデータは現在のディレクトリの *name*.profile/feedback に書き込まれます (*name* は実行ファイルの名前または-xprofile=collect:*name* フラグで指定された名前)。これらの変数が設定されると、-xprofile=collect データは $SUN_PROFDATA_DIR/$SUN_PROFDATA に書き込まれます。

$SUN_PROFDATA 環境変数と $SUN_PROFDATA_DIR 環境変数は、tcov によって書き込まれたプロファイルデータファイルのパスと名前を制御します。詳細は、tcov(1) マニュアルページを参照してください。

警告

別々の手順でコンパイルしてリンクする場合は、コンパイル手順とリンク手順で同じ -xprofile オプションを表示する必要があります。1 つの手順で -xprofile を取り込み、もう 1 つの手順で除外すると、プログラムの正確さは損なわれませんがプロファイルを行えなくなります。

関連項目

-xa、tcov(1) のマニュアルページ『プログラムのパフォーマンス解析』

-xprofile_ircache[=*path*]

collect 段階で保存されたコンパイルデータを再利用して use 段階のコンパイル時間を向上するには、-xprofile=collect|use で -xprofile_ircache[=*path*] を使用します。

中間データが保存されるので、大きいプログラムの場合には use フェーズのコンパイル時間を大幅に短縮できます。データが保存されるので、ディスク容量の必要量がかなり増えます。

-xprofile_ircache[=*path*] を使用すると、*path* はキャッシュファイルが保存されているディレクトリを上書きします。デフォルトでは、これらのファイルはオブジェクトファイルと同じディレクトリに保存されます。collect と use 段階が 2 つの別のディレクトリで実行される場合は、パスを指定しておくと便利です。一般的なコマンドシーケンスを次に示します。

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

-xprofile_pathmap

-xprofile=use コマンドも指定する場合は、-xprofile_pathmap=*collect_prefix*:*use_prefix* オプションを使用します。以下の項目がともに真で、コンパイラが -xprofile=use でコンパイルされたオブジェクトファイルのプロファイルデータを見つけられない場合は、-xprofile_pathmap を使用します。

- 前回オブジェクトファイルが、-xprofile=collect でコンパイルされたディレクトリとは異なるディレクトリで、オブジェクトファイルを -xprofile=use を指定してコンパイルしている。
- オブジェクトファイルはプロファイルで共通ベース名を共有しているが、異なるディレクトリのそれぞれの位置で相互に識別されている。

collect-prefix は、オブジェクトファイルが -xprofile=collect でコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

use-prefix は、オブジェクトファイルが -xprofile=use を指定してコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

-xprofile_pathmap の複数のインスタンスを指定すると、コンパイラは指定した順序でインスタンスを処理します。-xprofile_pathmap のインスタンスで指定された各 *use-prefix* は、一致する *use-prefix* が識別されるか、最後に指定された *use-prefix* がオブジェクトファイルのパス名と一致しないことが確認されるまで、オブジェクトファイルのパス名と比較されます。

-xregs=*r*[,*r*...]

SPARC: 一時レジスタの使用を制御します。

コンパイラは、一時記憶領域として使用できるレジスタ (一時レジスタ) が多ければ、それだけ高速なコードを生成します。このオプションは、利用できる一時レジスタを増やしますが、必ずしもそれが適切であるとは限りません。

値

r には次の値のいずれかを指定します。それぞれの値の意味は <c>-xarch</c> 設定によって異なります。

表 A-45 `-xregs` の値

r の値	意味
`[no%]appl`	コンパイラがアプリケーションレジスタをスクラッチレジスタとして使用してコードを生成することを許可します [しません]。アプリケーションレジスタは次のとおりです。 g2、g3、g4 (v8a、v8、v8plus、v8plusa、v8plusb) g2、g3 (v9、v9a、v9b) すべてのシステムソフトウェアおよびライブラリは、`-xreg=no%appl` を指定してコンパイルすることをお勧めします。システムソフトウェア (共有ライブラリを含む) は、アプリケーション用のレジスタの値を保持する必要があります。これらの値は、コンパイルシステムによって制御されるもので、アプリケーション全体で整合性が確保されている必要があります。 SPARC の命令セットの詳細については、326 ページの「`-xarch=isa`」を参照してください。 SPARC ABI では、これらのレジスタはアプリケーションレジスタと記述されています。これらのレジスタを使用すると、必要な `load` や `store` 命令が少なくなるため、パフォーマンスが向上します。ただし、これらのレジスタの使用は、他の目的でレジスタを使用するプログラムとの矛盾を起こすことがあります。
`[no%]float`	コンパイラが浮動小数点レジスタを整数値用のスクラッチレジスタとして使用してコードを生成することを許可します [しません]。浮動小数点値を使用する場合は、このオプションとは関係なくこれらのレジスタを使用します。`-xregs=no%float` を指定すると、ソースプログラムに浮動小数点コードを記述することはできません。

デフォルト

`-xregs` を指定しないと、`-xregs=appl,float` が使用されます。

例

使用可能なすべての一時レジスタを使ってアプリケーションプログラムをコンパイルするには、`-xregs=appl,float` と指定します。

コンテキストの切り替えの影響を受けやすい非浮動小数点コードをコンパイルするには`-xregs=no%appl,no%float` と指定します。

関連項目

SPARC V7/V8 ABI, SPARC V9 ABI

-xs

オブジェクト (`.o`) ファイルなしに `dbx` でデバッグできるようにします。

`-xs` オプションは、`dbx` の自動読み込みを無効にします。このオプションは、`.o` ファイルを残しておくことができない場合に使用してください。このオプションにより、`-s` オプションがアセンブラに渡されます。

「非自動読み込み」とは、シンボルテーブルの古い読み込み方法です。`dbx` の全シンボルテーブルが実行ファイル内に置かれます。その結果、リンカーによるリンクや `dbx` による初期化の速度が遅くなります。

「自動読み込み」は、シンボルテーブルの新しい読み込み方法 (デフォルト) です。各 `.o` ファイルに情報が含まれるため、`dbx` はシンボルテーブルが必要な場合にのみシンボルテーブル情報を読み込みます。このため、リンカーによるリンクや `dbx` による初期化の速度が速くなります。

`-xs` を指定する場合で、実行ファイルを別のディレクトリに移動して `dbx` を使用するときは、オブジェクト (`.o`) ファイルを移動する必要はありません。

`-xs` を指定せずに実行ファイルを別のディレクトリに移動して `dbx` を使用する場合は、ソースファイルとオブジェクト (`.o`) ファイルの両方を移動する必要があります。

-xsafe=mem

SPARC:メモリー保護違反が発生しなかったとコンパイラで想定されるようにすることができます。

このオプションを使用すると、コンパイラでは SPARC V6 アークテクチャーで違反のないロード命令を使用できます。

相互の関連性

このオプションは、`-xarch` で `v8plus`、`v8plusa`、`v8plusb`、`v9`、`v9a`、`v9b` のいずれかを指定し、最適化レベルの `-x05` と組み合わせた場合にだけ有効です。

警告

アドレスの位置合わせが合わない、またはセグメンテーション侵害などの違反が発生した場合は違反のないロードはトラップを引き起こさないので、このオプションはこのような違反が起こる可能性のないプログラムでしか使用しないでください。ほとんどのプログラムではメモリーに関するトラップは起こらないので、大多数のプログラムでこのオプションを安全に使用できます。メモリーに関するトラップを明示的に強制して例外条件を処理するプログラムの場合は、このオプションを使用しないでください。

-xsb

このオプションを指定すると、`CC` ドライバが、ソースブラウザのために `SunWS_cache` サブディレクトリにシンボルテーブル情報を追加生成します。

関連項目

`-xsbfast`

-xsbfast

ソースブラウザ情報を作成するだけでコンパイルはしません。

このオプションでは、`ccfe` 段階だけを実行して、ソースブラウザのために `SunWS_cache` サブディレクトリにシンボルテーブル情報を追加生成します。オブジェクトファイルは生成されません。

関連項目

`-xsb`

-xspace

SPARC:コードサイズが大きくなるような最適化は行いません。

-xtarget=*t*

命令セットと最適化処理の対象システムを指定します。

コンパイラにハードウェアシステムを正確に指定すると、プログラムによってはパフォーマンスが向上します。プログラムのパフォーマンスを重視する場合は、ハードウェアを適切に指定することが極めて重要です。特に、プログラムをより新しい SPARC システム上で実行する場合には重要になります。しかし、ほとんどのプログラムおよび旧式の SPARC システムではパフォーマンスの向上はわずかであるため、汎用的な指定方法で十分です。

`-xtarget` に指定する値は、`-xarch`、`-xchip`、`-xcache` の各オプションの値に展開されます。実行中のシステムで `-xtarget=native` の展開を調べるには、`fpversion`(1) コマンドを使用します。表 A-46を参照してください。

たとえば、`-xtarget=sun4/15` は次と同じです。`-xarch=v8a -xchip=micro -xcache=2/16/1.`

注 – 特定のホストプラットフォームで `-xtarget` を展開した場合、そのプラットフォームでコンパイルすると、`-xtarget=native` と同じ `-xarch`、`-xchip`、または `-xcache` 設定にならない場合があります。

値

SPARC プラットフォームの場合

SPARC プラットフォームでは、*t* には次のいずれかの値を指定します。

表 A-46 SPARC プラットフォームの`-xtarget`**の値**

t の値	意味
`native`	ホストシステムで最高のパフォーマンスが得られます。コンパイラは、ホストシステム用に最適化されたコードを生成します。コンパイラは自身が動作しているマシンで利用できるアーキテクチャ、チップ、キャッシュ特性を判定します。
`native64`	ホストシステムで 64 ビットのオブジェクトバイナリの最高のパフォーマンスが得られます。コンパイラは、ホストシステム用に最適化された 64 ビットのオブジェクトバイナリを生成します。コンパイラが動作しているマシンで使用できる 64 ビットのアーキテクチャ、チップ、キャッシュの属性を決定します。
`generic`	汎用アーキテクチャ、チップ、キャッシュで最高のパフォーマンスが得られます。 コンパイラは `-xtarget=generic` を次のように展開します。 `-xarch=generic -xchip=generic -xcache=generic.` これはデフォルト値です。
`generic64`	大多数の 64 ビットのプラットフォームのアーキテクチャーで 64 ビットのオブジェクトバイナリの最適なパフォーマンスを得るためのパラメータを設定します。
platform-name	指定するシステムで最高のパフォーマンスが得られます。表 A-47 から SPARC プラットフォームの名前を選択します。

次の表は、`-xtarget` に指定できる SPARC プラットフォームの名前とその展開値を示しています。

表 A-47 `-xtarget` **の SPARC プラットフォーム名**

-xtarget=	-xarch	-xchip	-xcache
`generic`	`generic`	`generic`	`generic`
`cs6400`	`v8plusa`	`super`	`16/32/4:2048/64/1`
`entr150`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2`	`v8plusa`	`ultra`	`16/32/1:512/64/1`
`entr2/1170`	`v8plusa`	`ultra`	`16/32/1:512/64/1`

表 A-47 -xtarget の **SPARC** プラットフォーム名 **(**続き**)**

-xtarget=	-xarch	-xchip	-xcache
entr2/1200	v8plusa	ultra	16/32/1:512/64/1
entr2/2170	v8plusa	ultra	16/32/1:512/64/1
entr2/2200	v8plusa	ultra	16/32/1:512/64/1
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8plusa	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1

表 A-47 -xtarget の **SPARC** プラットフォーム名 **(**続き**)**

-xtarget=	-xarch	-xchip	-xcache
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1

表 A-47 -xtarget の **SPARC** プラットフォーム名 **(**続き**)**

-xtarget=	-xarch	-xchip	-xcache
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1

表 A-47 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget=	-xarch	-xchip	-xcache
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/512/1
ultra3cu	v8plusa	ultra3cu	64/32/4:8192/512/2

IA プラットフォームの場合

IA プラットフォームでは、t に次の値を指定できます。

表 A-48 **IA** プラットフォームの `-xtarget`の値

t の値	意味
`generic`	汎用アーキテクチャ、チップ、キャッシュで最高のパフォーマンスが得られます。 これはデフォルト値です。
`native`	ホストシステムで最高のパフォーマンスが得られます。
`386`	Intel 80386 マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
`486`	Intel 80486 マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
`pentium`	Pentium マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。
`pentium_pro`	Pentium Pro マイクロプロセッサで最高のパフォーマンスが得られるコードが生成されます。

次の表は、Intel アーキテクチャでの `-xtarget` の値を示しています。

表 A-49 **Intel** アーキテクチャでの `-xtarget` の展開

-xtarget=	-xarch	-xchip	-xcache
`generic`	`generic`	`generic`	`generic`
`386`	`386`	`386`	`generic`
`486`	`386`	`486`	`generic`
`pentium`	`386`	`pentium`	`generic`
`pentium_pro`	`pentium_pro`	`pentium_pro`	`generic`

デフォルト

SPARC でも IA でも、`-xtarget` を指定しないと、`-xtarget=generic` が使用されます。

拡張

-xtarget オプションは、購入したプラットフォーム上で使用する -xarch、-xchip、-xcache の組み合わせを簡単に指定するためのマクロです。-xtarget の意味は = の後に指定した値を展開したものにあります。

例

-xtarget=sun4/15 は -xarch=v8a -xchip=micro -xcache=2/16/1 を意味します。

相互の関連性

-xarch=v9|v9a|v9b オプションで指定する、SPARC V9 アーキテクチャのコンパイル。-xtarget=ultra や ultra2 の設定は、必要でないか、十分ではありません。-xtarget を指定する場合は、-xarch=v9|v9a|v9b オプションは -xtarget よりも後になければなりません。例を次に示します。

```
-xarch=v9 -xtarget=ultra
```

上記の指定は次のように展開され、-xarch の値が v8 に戻ります。

```
-xarch=v9 -xarch=v8 -xchip=ultra -xcache=16/32/1:512/64/1
```

正しくは、次のように、-xarch を -xtarget よりも後に指定します。例を次に示します。

```
-xtarget=ultra -xarch=v9
```

警告

別々の手順でコンパイルしてリンクする場合は、コンパイル手順とリンク手順で同じ -xtarget の設定値を使用する必要があります。

-xthreadvar[=*o*]

(SPARC) スレッドローカルな変数の実装を制御するには -xthreadvar を指定します。コンパイラのスレッドローカルな記憶機能を利用するには、このオプションを __thread 宣言指定子と組み合わせて使用します。__thread 指定子を使用してスレッド変数を宣言した後は、-xthreadvar を指定して動的 (共有) ライブラリの位置依存コード (非 PIC コード) でスレッドローカルな記憶領域を使用できるようにします。__thread の使用方法の詳細については、51 ページの「スレッドローカルな記憶装置」を参照してください。

値

o には、以下のいずれかを指定します。

表 A-50 -xthreadvar の値

r の値	意味
[no%]dynamic	動的ロード用の変数をコンパイルします [しません]。-xthreadvar=no%dynamic を指定すると、スレッド変数へのアクセスは非常に速くなりますが、動的ライブラリ内のオブジェクトファイルは使用できません。すなわち、実行可能ファイル内のオブジェクトファイルは使用できません。

デフォルト

-xthreadvar を指定しない場合、コンパイラが使用するデフォルトは位置独立コードが有効になっているかどうかによって決まります。位置独立コードが有効になっている場合、オプションは -xthreadvar=dynamic に設定されます。位置独立コードが無効になっている場合、オプションは -xthreadvar=no%dynamic に設定されます。

引数を指定しないで -xthreadva を指定する場合、オプションは -xthreadvar-dynamic に設定されます。

相互の関連性

異なるバージョンの Solaris ソフトウェアでスレッド変数を使用するには、コマンド行で異なるオプションを指定する必要があります。

- Solaris 8 ソフトウェアでは、__thread を使用するオブジェクトは -mt を指定してコンパイルし、-mt -L/usr/lib/lwp -R/usr/lib/lwp を指定してリンクする必要があります。
- Solaris 9 ソフトウェアでは、__thread を使用するオブジェクトはコンパイルとリンクに -mt を指定する必要があります。

警告

動的ライブラリ内に位置依存コード (非 PIC コード) がある場合、-xthreadvar を指定する必要があります。

リンカーは、動的ライブラリ内の 非 PIC コードと同等のスレッド変数はサポートできません。非 PIC スレッド変数は非常に速いので、実行可能ファイルのデフォルトとして指定できます。

関連項目

-xcode、-KPIC、-Kpic

-xtime

cc ドライバが、さまざまなコンパイル過程の実行時間を報告します。

-xtrigraphs[={yes|no}]

ISO/ANSI C 標準の定義に従って文字表記シーケンスの認識を有効または無効にします。

コンパイラが文字表記シーケンスとして解釈している疑問符 (?) の入ったリテラル文字列がソースコードにある場合は、-xtrigraph=no サブオプションを使用して文字表記シーケンスの認識をオフにすることができます。

値

-xtrigraphs には、以下のいずれかを指定します。

表 A-51 -xtrigraphs の値

値	意味
yes	コンパイル単位全体の 3 文字表記の認識を有効にします。
no	コンパイル単位全体の 3 文字表記の認識を無効にします。

デフォルト

コマンド行に -xtrigraphs オプションを指定しなかった場合、コンパイラは -xtrigraphs=yes を使用します。

-xtrigraphs だけを指定すると、コンパイラは -xtrigraphs=yes を使用します。

例

trigraphs_demo.cc という名前のソースファイル例を参照してください。

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

このコードに -xtrigraphs=yes を指定してコンパイルした場合の出力は次のとおりです。

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as (]
```

このコードに -xtrigraphs=no を指定してコンパイルした場合の出力は次のとおりです。

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

関連項目

3 文字表記については、『C ユーザーズガイド』の ANSI/ISO C への移行に関する章を参照してください。

-xunroll=*n*

可能な場合は、ループを展開します。

このオプションは、コンパイラがループを最適化 (展開) するかどうかを指定します。

値

n が 1 の場合、コンパイラはループを展開しません。

n が 1 より大きな整数の場合は、`-unroll=`*n* によってコンパイラがループを *n* 回展開します。

-xustr={ascii_utf16_ushort|no}

コンパイラにオブジェクトファイル内で UTF-16 文字列に変換させたい文字列リテラルがコードに含まれる場合、このオプションを使用します。このオプションが指定されていない場合、コンパイラは 16 文字の文字列リテラルを出力することも認識することもしません。このオプションを使用すれば、U"*ASCII_string*" 文字列リテラルが unsigned short int の配列として認識されます。この種の文字列が含まれている標準は現時点ではまだないため、このオプションは、非標準 C++ の認識を可能とします。

すべてのファイルを、このオプションによってコンパイルしなければならないわけではありません。

値

ISO10646 UTF-16 文字列リテラルを使用する国際化アプリケーションをサポートする必要がある場合には、`-xustr=ascii_utf16_ushort` を指定します。`-xustr=no` を指定すれば、コンパイラが U"*ASCII_string*" 文字列リテラルを認識しなくなります。このオプションのコマンド行の右端にあるインスタンスは、それまでのインスタンスをすべて上書きします。

`-xustr=ascii_ustf16_ushort` は、U"*ASCII_string*" 文字列リテラルを指定しなくてもかまいません。そのようにしても、エラーとはなりません。

デフォルト

デフォルトは -xustr=no です。引数を指定しないで -xustr を指定した場合、コンパイラはこの指定を受け付けず、警告を出力します。C または C++ の標準で構文の意味が定義されれば、デフォルト値を変更できます。

例

次の例は、U が先頭に付いた、引用符に入った文字列リテラルを示しています。-xustr を指定するコマンドも示しています。

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() { return U"fun" };
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

警告

16 ビットの文字リテラルはサポートされていません。

-xvis[={yes|no}]

(SPARC) VIS™ instruction-set Software Developers Kit (VSDK) に定義されているアセンブリ言語のテンプレートを使用する場合は、-xvis=[yes|no] コマンドを使用します。

VIS 命令セットは、SPARC v9 命令セットの拡張機能です。UltraSPARC プロセッサは 64 ビットですが、データサイズが 8 ビットや 16 ビットに限定されていることが少なくなく、マルチメディアアプリケーションではとくによく見られます。VIS 命令は 1 つの命令で 16 ビットデータを 4 個処理できるため、イメージング、線形代数、シグナル処理、オーディオ、ビデオ、ネットワーキングといった新しいメディアを扱うアプリケーションのパフォーマンスが大きく向上します。

デフォルト

デフォルトは、-xvis=no です。-xvis と指定すると -xvis=yes と指定した場合と同様の結果が得られます。

関連項目

VSDK の詳細については、`http://www.sun.com/processors/vis/` を参照してください。

`-xwe`

ゼロ以外の終了状態を返すことによって、すべての警告をエラーとして扱います。

`-z[ ]`*arg*

リンクエディタのオプション。詳細は、`ld`(1) のマニュアルページと Solaris 関連のマニュアル『リンカーとライブラリ』を参照してください。

付録B

プラグマ

この章では、プラグマについて説明します。「プラグマ」とは、コンパイラに特定の情報を渡すために使用するコンパイラ指令です。プラグマを使用すると、コンパイル内容を詳細に渡って制御できます。たとえば、pack プラグマを使用すると、構造体の中のデータの配置を変えることができます。プラグマは「指令」とも呼ばれます。

プリプロセッサキーワードの pragma は、C++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。プラグマは C++ 標準には定義されていません。

注 – したがってプラグマに依存するコードには移植性はありません。プラグマに依存するコードは移植性がありません。

プラグマの書式

次に、C++ コンパイラのプラグマのさまざまな書式を示します。

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

変数 *keyword* は特定の指令を示し、*a* は引数を示します。

プラグマの詳細

この節では、Sun WorkShop C++ コンパイラが認識するプラグマキーワードについて説明します。

- `align`

デフォルトを無効にして、パラメータ変数のメモリー境界を、指定したバイト境界に揃えます。

- `does_not_read_global_data`

指定リストの関数が、大域データを直接的にも間接的にも読み込まないことをコンパイラに宣言します。

- `does_not_return`

指定関数の呼び出しが返らないことをコンパイラに宣言します。

- `does_not_write_global_data`

指定リストの関数が、大域データを直接的にも間接的にも書き込まないことをコンパイラに宣言します。

- `dump_macros`

コード内でのマクロの使用方法を説明します。

- `end_dumpmacros`

`dump_macros` プラグマの終わりであることを記します。

- `fini`

指定した関数を終了関数にします。

- `hdrstop`

プリコンパイル済みヘッダーの活性文字列の終わりを示します。

- `ident`

実行可能ファイルの `.comment` 部に、指定した文字列を入れます。

- `init`

指定した関数を初期化関数にします。

- `no_side_effect`

永続性を持つ状態が関数によって変更されないことを通知します。

- `pack` (*n*)

構造体オフセットの配置を制御します。*n* の値は、すべての構造体メンバーに合った最悪の場合の境界整列を指定する数字で、0、1、2、4、8 のいずれかにします。

- `rarely_called`

指定した関数が呼び出されることはほとんどないことをコンパイラに通知します。

- `returns_new_memory`

指定した関数が新しく割り当てられたメモリーのアドレスを返し、そのポインタが他のポインタの別名として使用されないことをコンパイラに宣言します。

- `unknown_control_flow`

手続き呼び出しの通常の制御フロー属性に違反するルーチンのリストを指定します。

- `weak`

弱いシンボル結合を定義します。

#pragma align

```
#pragma align integer(variable [,variable...])
```

`align` を使用すると、指定したすべての変数 *variable* (変数) のメモリー境界を *integer* (整数) バイト境界に揃えることができます (デフォルト値より優先されます) 。ただし、次の制限があります。

- *integer* は、1 ~ 128 の範囲にある 2 の二乗、つまり、1、2、4、8、16、32、64、128 のいずれかでなければなりません。
- *variable* には、大域変数か静的変数を指定します。局所変数またはクラスメンバー変数は指定できません。

- 指定されたメモリーの境界値がデフォルト値より小さいと、デフォルト値が使用されます。
- この #pragma 行は、指定した変数の宣言より前になければなりません。前にないと、この #pragma 行は無視されます。
- この #pragma 行で指定されていても、プラグマ行に続くコードの中で宣言されない変数は、すべて無視されます。次に、正しく宣言されている例を示します。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

#pragma align を名前空間内で使用するときは、符号化された名前を使用する必要があります。たとえば、次のコード中の、#pragma align 文には何の効果もありません。この問題を解決するには、#pragma align 文の a、b、および c を符号化された名前に変更します。

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

#pragma does_not_read_global_data

```
#pragma does_not_read_global_data(funcname [, funcname])
```

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを読み込まないことをコンパイラに宣言します。この結果、こういったルーチンの呼び出しの周辺のコードをより良く最適化できます。とくに、代入文や代入ストアの場合は、こういった呼び出しのまわりを移動させることができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言した後に限定されます。大域アクセスに関する宣言が真でない場合、プログラムの動作は未定義となります。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

#pragma does_not_return

```
#pragma does_not_return(funcname [, funcname])
```

このプラグマは、指定のルーチンの呼び出しが返らないことをコンパイラに宣言します。この結果、コンパイラは、この宣言に整合した最適化を行えます。たとえば、呼び出し側で終了したライフ回数を登録すると、より多くの最適化が可能となります。

指定の関数が返らない場合、プログラムの動作は未定義となります。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

次の例のとおり、このプラグマを使用できるのは、指定した関数のプロトタイプを宣言した後に限定されます。

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

#pragma does_not_write_global_data

```
#pragma does_not_wrtie_global_data(funcname [, funcname])
```

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを書き込まないことをコンパイラに宣言します。この結果、こういったルーチンの呼び出しの周辺のコードをより良く最適化できます。とくに、代入文や代入ストアの場合は、こういった呼び出しのまわりを移動させることができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言した後に限定されます。大域アクセスに関する宣言が真でない場合、プログラムの動作は未定義となります。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

#pragma dumpmacros

```
#pragma dumpmacros (value[,value...])
```

マクロがプログラム内でどのように動作しているかを調べたいときに、このプラグマを使用します。このプラグマは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に従って、標準エラー (stderr) に出力します。dumpmacros プラグマは、ファイルが終わるまで、または #pragma end_dumpmacro に到達するまで、有効です。427 ページの「#pragma end_dumpmacros」を参照してください。*value* の代わりに次の引数を使用できます。

値	内容
defs	すべての定義済みマクロを出力します
undefs	すべての解除済みマクロを出力します
use	使用されているマクロの情報を出力します
loc	defs、undefs、use の位置 (パス名と行番号) も出力します
conds	条件付き指令で使用したマクロの使用情報を出力します
sys	システムヘッダーファイルのマクロについて、すべての定義済みマクロ、解除済みマクロ、使用状況も出力します。

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果はありません。たとえば #pragma dumpmacros=loc,conds,sys には、何も効果はありません。

dumpmacros プラグマとコマンド行オプションの効果は同じですが、プラグマがコマンド行オプションを上書きします。344 ページの「-xdumpmacros[=value[,value...]]」を参照してください。

dumpmacros プラグマは入れ子にならないので、次のコードでは、#pragma end_dumpmacros が処理されるとマクロ情報の出力が停止します。

```
#pragma dumpmacros (defs, undefs)
#pragma dumpmacros (defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros プラグマの効果は累積的です。以下は、

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

以下と同じ効果を持ちます。

```
#pragma dumpmacros(defs, undefs, loc)
```

オプション #pragma dumpmacros=use,no%loc を使用した場合、使用したマクロそれぞれの名前が一度だけ出力されます。オプション #pragma dumpmacros=use,loc を使用した場合、マクロを使用するたびに位置とマクロ名が出力されます。

#pragma end_dumpmacros

```
#pragma end_dumpmacros
```

このプラグマは、dumpmacros pragma が終わったことを通知し、マクロ情報の出力を停止します。dumpmacros プラグマ終了時に end_dumpmacros プラグマを使用しなかった場合、dumpmacros プラグマはファイルが終わるまで出力を生成し続けます。

#pragma fini

```
#pragma fini (identifier [, identifier ...])
```

fini を使用すると、*identifier* (識別子) を「終了関数」にします。この関数は void 型で、引数を持ちません。この関数は、プログラム制御によってプログラムが終了する時、または関数内の共有オブジェクトがメモリーから削除されるときに呼び出されます。初期設定関数と同様に、終了関数はリンカーが処理した順序で実行されます。

ソースファイル内で#pragma finiで指定された関数は、そのファイルの中にある静的デストラクタの後に実行されます。*identifier* は、この で指定する前に宣言しておく必要があります。

#pragma hdrstop

hdrstop プラグマをソースファイルヘッダーに埋め込むと、活性文字列の終わりが指示されます。たとえば次のファイルがあるとします。

```
example% cat a.cc
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

活性文字列は c.h で終わるので、各ファイルのc.h の後に #pragma hdrstop を挿入します。

#pragma hdrstop を挿入できる場所は、CC コマンドで指定したソースファイルの活性文字列の終わりだけです。#pragma hdrstop をインクルードファイル内に指定しないでください。

383 ページの「-xpch=v」 と 387 ページの「-xpchstop=file」 を参照してください。

#pragma ident

```
#pragma ident string
```

ident を使用すると、実行可能ファイルの .comment 部に、*string* に指定した文字列を記述することができます。

#pragma init

```
#pragma init(identifier[,identifier...])
```

init を使用すると、*identifier* (識別子) を「初期設定関数」にします。この関数は void 型で、引数を持ちません。この関数は、実行開始時にプログラムのメモリーイメージを構築する時に呼び出されます。共有オブジェクトの初期設定子の場合、共有オブジェクトをメモリーに入れるとき、つまりプログラムの起動時または dlopen() のような動的ロード時のいずれかに実行されます。初期設定関数の呼び出し順序は、静的と動的のどちらの場合でもリンカーが処理した順序になります。

ソースファイル内で #pragma init で指定された関数は、そのファイルの中にある静的コンストラクタの後に実行されます。*identifier* は、この#pragma で指定する前に宣言しておく必要があります。

#pragma no_side_effect

```
#pragma no_side_effect(name[,name...])
```

no_side_effect は、関数によって持続性を持つ状態が変更されないことを通知するためのものです。このプラグマは、指定された関数がどのような副作用も起こさないことをコンパイラに宣言します。すなわち、これらの関数は、渡された引数だけに依存する値を返します。さらに、これらの関数と、そこから呼び出される関数は、次の処理も行ってはなりません。

- 呼び出された時点で、呼び出し元から参照可能になっているプログラム状態の読み取りや書き込み
- 入出力の実行

■ 呼び出された時点で参照可能になっていないプログラム状態の変更

コンパイラは、この情報を最適化に使用します。

このプラグマで指定した関数が、実際には副作用を与える場合は、この関数を呼び出したプログラムの実行結果は保証されません。

name 引数で指定する関数は、現在の翻訳単位に含まれていなければなりません。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければなりません。プラグマは、関数定義の前に位置していなければなりません。

該当する関数を多重定義している場合は、このプラグマは最後に定義した関数に適用されます。その関数を別の識別名を使用して定義している場合は、このプラグマはエラーになります。

#pragma pack(*n*)

```
#pragma pack([n])
```

pack は、構造体メンバーの配置制御に使用します。

n を指定する場合、0 であるか 2 の累乗でなければなりません。0 以外の値を指定すると、コンパイラは *n* バイトの境界整列と、データ型に対するプラットフォームの自然境界のどちらか小さい方を使用します。たとえば次の指令は、自然境界整列が 4 バイトまたは 8 バイト境界である場合でも、指令の後 (および後続の pack 指令の前) に定義されているすべての構造体のメンバーを 2 バイト境界を超えないように揃えます。

```
#pragma pack(2)
```

n が 0 であるか省略された場合、メンバー整列は自然境界整列の値に戻ります。

n の値がプラットフォームのもっとも厳密な境界整列と同じかそれ以上の場合には、自然境界整列になります。次の表に、各プラットフォームのもっとも厳密な境界整列を示します。

表 B-1 プラットフォームのもっとも厳密な境界整列

プラットフォーム	もっとも厳密な境界整列
IA	4
SPARC 一般、V7、V8、V8a、V8plus、V8plusa、V8plusb	8
SPARC V9, V9a, V9b	16

`pack` 指令は、次の `pack` 指令までに存在するすべての構造体定義に適用されます。別々の翻訳単位にある同じ構造体に対して異なる境界整列が指定されると、プログラムは予測できない状態で異常終了する場合があります。特に、コンパイル済みライブラリのインタフェースを定義するヘッダーをインクルードする場合は、その前に `pack` を使用しないでください。プログラムコード内では、`pack` 指令は境界整列を指定する構造体の直前に置き、`#pragma pack ()` は構造体の直後に置くことをお勧めします。

SPARC プラットフォーム上で `#pragma pack` を使用して、型のデフォルトの境界整列よりも密に配置するには、アプリケーションのコンパイルとリンクの両方で `-misalign` オプションを指定する必要があります。次の表に、整数データ型のメモリーサイズとデフォルトの境界整列を示します。

表 B-2 メモリーサイズとデフォルトの境界整列 **(**単位はバイト数**)**

型	SPARC V8 サイズ、境界整列	SPARC V9 サイズ、境界整列	IA サイズ、境界整列
bool	1, 1	1, 1	1, 1
char	1, 1	1, 1	1, 1
short	2, 2	2, 2	2, 2
wchar_t	4, 4	4, 4	4, 4
int	4, 4	4, 4	4, 4
long	4, 4	8, 8	4, 4

表 B-2　メモリーサイズとデフォルトの境界整列 (単位はバイト数) (続き)

型	SPARC V8 サイズ、境界整列	SPARC V9 サイズ、境界整列	IA サイズ、境界整列
float	4, 4	4, 4	4, 4
double	8, 8	8, 8	8, 4
long double	16, 8	16, 16	12, 4
pointer to data	4, 4	8, 8	4, 4
pointer to function	4, 4	8, 8	4, 4
pointer to member data	4, 4	8, 8	4, 4
pointer to member function	8, 4	16, 8	8, 4

#pragma rarely_called

```
#pragms rarely_called(funcname[, funcname])
```

このプラグマは、指定の関数がほとんど呼び出されないことをコンパイラに示唆します。この結果、こういったルーチンの呼び出し側に対するプロファイル-フィードバック形式の最適化を、プロファイルの収集を行わずにコンパイラが実施できます。このプラグマが示すことは示唆に過ぎないので、このプラグマに基づいた最適化をコンパイラが行わないこともあります。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

#pragma rarely_called プリプロセッサ指令を使用できるのは、指定の関数のプロトタイプが宣言された後だけです。以下は、#pragma rarely_called の例です。

```
extern void error (char *message);
#pragma rarely_called(error)
```

#pragma returns_new_memory

```
#pragma returns_new_memory(name[,name...])
```

このプラグマは、指定した関数が新しく割り当てられたメモリーのアドレスを返し、そのポインタが他のポインタの別名として使用されないことをコンパイラに宣言します。この情報により、オプティマイザはポインタ値をより正確に追跡し、メモリー位置を明確化することができます。この結果、スケジューリングとパイプライン化が改善されます。

このプラグマの宣言が実際には誤っている場合は、該当する関数を呼び出したプログラムの実行結果は保証されません。

name 引数で指定する関数は、現在の翻訳単位に含まれていなければなりません。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければなりません。プラグマは、関数定義の前に位置していなければなりません。

該当する関数を多重定義している場合は、このプラグマは最後に定義した関数に適用されます。その関数を別の識別名を使用して定義している場合は、このプラグマはエラーになります。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

#pragma unknown_control_flow

```
#pragma unknown_control_flow(name[,name...])
```

unknown_control_flow を使用すると、手続き呼び出しの通常の制御フロー属性に違反するルーチンの名前のリスト[,*name*,[,*name*]... を指定できます。たとえば、setjmp()の直後の文は、他のどんなルーチンを呼び出してもそこから返ってくることができます。これは、longjmp() を呼び出すことによって行います。

このようなルーチンを使用すると標準のフローグラフ解析ができないため、呼び出す側のルーチンを最適化すると安全性が確保できません。このような場合に #pragma unknown_control_flow を使用すると安全な最適化が行えます。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

#pragma weak

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

weak を使用すると、弱い (weak) 大域シンボルを定義できます。このプラグマは主にソースファイルの中でライブラリを構築するために使用されます。リンカーは弱いシンボルを認識できなくてもエラーメッセージを出しません。

weak プラグマは、次の 2 つの書式でシンボルを指定できます。

- **文字列**文字列は、C++ の変数または関数の符号化された名前でなければなりません。無効な符号化名が指定された場合、その名前を参照したときの動作は予測できません。無効な符号化名を参照した場合、バックエンドがエラーを生成するかどうかは不明です。エラーを生成するかどうかに関わらず、無効な符号化名を参照したときのバックエンドの動作は予測できません。
- **識別子**識別子は、コンパイル単位内であらかじめ宣言された C++ の関数のあいまいでない識別子でなければなりません。識別子書式は変数には使用できません。無効な識別子への参照を検出した場合、フロントエンド (ccfe) はエラーメッセージを生成します。

#pragma weak *name*

#pragma weak *name* という書式の指令は、*name* を弱い (weak) シンボルに定義します。*name* のシンボル定義が見つからなくても、リンカーはエラーメッセージを生成しません。また、弱いシンボルの定義を複数見つけた場合でも、リンカーはエラーメッセージを生成しません。リンカーは単に最初に検出した定義を使用します。

プラグマ *name* の強い定義が存在しない場合、リンカーはシンボルの値を 0 にします。

次の指令は、ping を弱いシンボルに定義しています。ping という名前のシンボルの定義が見つからない場合でも、リンカーはエラーメッセージを生成しません。

```
#pragma weak ping
```

#pragma weak *name1* = *name2*

#pragma weak *name1* = *name2* という書式の指令は、シンボル *name1* を *name2* への弱い参照として定義します。*name1* がどこにも定義されていない場合、*name1* の値は *name2* の値になります。*name1* が別の場所で定義されている場合、リンカーはその定

義を使用し、*name2* への弱い参照は無視します。次の指令では、bar がプログラムのどこかで定義されている場合、リンカーはすべての参照先を bar に設定します。そうでない場合、リンカーは bar への参照を foo にリンクします。

```
#pragma weak bar = foo
```

識別子書式では、*name2* は現在のコンパイル単位内で宣言および定義しなければなりません。例を次に示します。

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

文字列書式を使用する場合、シンボルはあらかじめ宣言されている必要はありません。次の例において、_bar と bar の両方が extern "C" である場合、その関数はあらかじめ宣言されている必要はありません。しかし、bar は同じオブジェクト内で定義されている必要があります。

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

関数の多重定義

識別子書式を使用するとき、プラグマのあるスコープ中には指定した名前を持つ関数は、1 つしか存在してはなりません。多重定義された関数を使用して #pragma weak の識別子書式を使用しようとするとエラーになります。次に例を示します。

```
int bar(int);
float bar(float);
#pragma weak bar        //あいまいな関数名により、エラー
```

このエラーを回避するには、文字列書式を使用します。次に例を示します。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

詳細は、『リンカーとライブラリ』を参照してください。

用語集

ABI

「アプリケーションバイナリインタフェース」を参照。

ANSI C

ANSI (米国規格協会) による C プログラミング言語の定義。ISO (国際標準化機構) 定義と同じです。「ISO」を参照。

ANSI/ISO C++

米国規格協会と国際標準化機構が共同で作成した C++ プログラミング言語の標準。「ISO」を参照。

cfront

C++ を C ソースコードに変換する C++ から C へのコンパイルプログラム。変換後の C ソースコードは、標準の C コンパイラでコンパイルできます。

ISO

国際標準化機構。

K&R C

Brian Kernighan と Dennis Ritchie によって開発された、ANSI C 以前の事実上の C プログラミング言語標準。

VTABLE

仮想関数を持つクラスごとにコンパイラが作成するテーブル。

アプリケーションバイナリインタフェース

コンパイルされたアプリケーションとそのアプリケーションが動作するオペレーティングシステム間のバイナリシステムインタフェース。

インクリメンタルリンカー

変更された .o ファイルだけを古い実行可能ファイルにリンクして新しい実行可能ファイルを作成するリンカー。

インスタンス化

C++ コンパイラが、テンプレートから使用可能な関数やオブジェクト (インスタンス) を生成する処理。

インスタンス変数

特定のオブジェクトに関連付けられたデータ項目。クラスの各インスタンスは、クラス内で定義されたインスタンス変数の独自のコピーを持っています。フィールドとも呼びます。「クラス変数」も参照。

インライン関数

関数呼び出しを実際の関数コードに置き換える関数。

右辺値

代入演算子の右辺にある変数。右辺値は読み取れますが、変更はできません。

演算子の多重定義

同じ演算子表記を異なる種類の計算に使用できること。関数の多重定義の特殊な形式の 1 つです。

オプション

「コンパイラオプション」を参照。

型

シンボルをどのように使用するかを記述したもの。基本型は整数と浮動小数点数であり、他のすべての型は、これらの基本型を配列や構造体にしたり、ポインタ属性や定数属性などの修飾子を加えることによって作成されます。

関数の多重定義

扱う引数の型と個数が異なる複数の関数に、同じ名前を与えること。関数の多相性ともいいます。

関数の多相性

「関数の多重定義」を参照。

関数のテンプレート

ある関数を作成し、それを「ひな型」として関連する関数を作成するための仕組み。

関数プロトタイプ

関数とプログラムの残りの部分とのインタフェースを記述する宣言。

キーワード

プログラミング言語で固有の意味を持ち、その言語において特殊な文脈だけで使用可能な単語。

基底クラス

「継承」を参照。

局所変数

ブロック内のコードからはアクセスできるが、ブロック外のコードからはアクセスできないデータ項目。たとえば、メソッド内で定義された変数は局所変数であり、メソッドの外からは使用できません。

クラス

名前が付いた一連のデータ要素 (型が異なっていてもよい) と、そのデータを処理する一連の演算からなるユーザーの定義するデータ型。

クラステンプレート

一連のクラスや関連するデータ型を記述したテンプレート。

クラス変数

クラスの特定のインスタンスではなく、特定のクラス全体を対象として関連付けられたデータ項目。クラス変数はクラス定義中に定義されます。静的フィールドとも呼びます。「インスタンス変数」も参照。

継承

プログラマが既存のクラス (基底クラス) から新しいクラス (派生クラス) を派生させることを可能にするオブジェクト指向プログラミングの機能。継承の種類には、公開、限定公開、非公開があります。

コンストラクタ

クラスオブジェクトを作成するときにコンパイラによって自動的に呼び出される特別なクラスメンバー関数。これによって、オブジェクトのインスタンス変数が初期化されます。コンストラクタの名前は、それが属するクラスの名前と同じでなければなりません。「デストラクタ」を参照。

コンパイラオプション

コンパイラの動作を変更するためにコンパイラに与える命令。たとえば、-g オプションを指定すると、デバッガ用のデータが生成されます。フラグやスイッチとも呼ばれます。

最適化

コンパイラが生成するオブジェクトコードの効率を良くする処理のこと。

サブルーチン

関数のこと。Fortran では、値を返さない関数を指します。

左辺値

変数のデータ値が格納されているメモリーの場所を表す式。あるいは、代入演算子の左辺にある変数のインスタンス。

事前束縛

「静的束縛」を参照。

実行時型識別機構 (RTTI)

プログラムが実行時にオブジェクトの型を識別できるようにする標準的な方法を提供する仕組み。

実行時束縛

「動的束縛」を参照。

シンボル

何らかのプログラムエントリを示す名前やラベル。

シンボルテーブル

プログラムのコンパイルで検出されたすべての識別子と、それらのプログラム中の位置と属性からなるリスト。コンパイラは、このテーブルを使って識別子の使い方を判断します。

スイッチ

「コンパイラオプション」を参照。

スコープ

あるアクションまたは定義が適用される範囲。

スタック

後入れ先出し法によってデータをスタックの一番上に追加するか、一番上から削除しなければならないデータ記憶方式。

スタブ

オブジェクトコードに生成されるシンボルテーブルのエントリ。デバッグ情報を含む `a.out` ファイルと ELF ファイルには同じ形式のスタブが使用されます。

静的束縛

関数呼び出しと関数本体をコンパイル時に結び付けること。事前束縛とも呼びます。

束縛

関数呼び出しを特定の関数定義に関連付けること。一般的には、名前を特定のエントリに関連付けることを指します。

多重継承

複数の基底クラスから 1 つの派生クラスを直接継承すること。

多重定義

複数の関数や演算子に同じ名前を指定すること。

多相性

ポインタや参照が、自分自身の宣言された型とは異なる動的な型を持つオブジェクトを参照できること。

抽象クラス

1 つまたは複数の抽象メソッドを持つクラス。したがって、抽象クラスはインスタンス化できません。抽象クラスは、他のクラスが抽象クラスを拡張し、その抽象メソッドを実装することで具体化されることを目的として、定義されています。

抽象メソッド

実装を持たないメソッド。

データ型

文字、整数、浮動小数点数などを表現するための仕組み。変数に割り当てられる記憶域とその変数に対してどのような演算が実行可能かは、この型によって決まります。

データメンバー

クラスの要素であるデータ。関数や型定義と区別してこのように呼ばれます。

デストラクタ

クラスオブジェクトを破棄したり、演算子 `delete` をクラスポインタに適用したときにコンパイラによって自動的に呼び出される特別なクラスメンバー関数。デストラクタの名前は、それが属するクラスの名前と同じで、かつ、名前の前にチルド (~) が必要です。「コンストラクタ」を参照。

テンプレートオプションファイル

テンプレートのコンパイル用オプションやソースの位置などの情報が含まれている、ユーザーが用意するファイル。テンプレートオプションファイルの使用は推奨されていないため、使用すべきではありません。

テンプレートデータベース

プログラムが必要とするテンプレートの処理とインスタンス化に必要なすべての構成ファイルを含むディレクトリ。

テンプレートの特殊化

デフォルトのインスタンス化では型を適切に処理できないときに、このデフォルトを置き換える、クラステンプレートメンバー関数の特殊インスタンス。

動的キャスト

ポインタや参照の型を、宣言されたものから、それが参照する動的な型と矛盾しない任意の型に安全に変換するための方法。

動的束縛

関数呼び出しと関数本体を実行時に結び付けること。これは、仮想関数に対してのみ行われます。事後束縛または実行時束縛とも呼ばれます。

動的な型

ポインタや参照でアクセスするオブジェクトの実際の型。この型は、宣言された型と異なることがあります。

トラップ

他の処置をとるためにプログラムの実行などの処置を遮ること。これによって、マイクロプロセッサの演算が一時的に中断され、プログラム制御が他のソースに渡されます。

名前空間

大域空間を一意の名前を持つスコープに分割して、大域的な名前のスコープを制御する仕組み。

名前の符号化

C++ では多くの関数が同じ名前を持つことがあるため、名前だけでは関数を区別できません。そこで、コンパイラは関数名とパラメータを組み合わせた一意の名前を各関数に割り当てます。このことを名前の符号化と呼びます。これによって、型の誤りのないリンケージを行うことができます。名前の符号化は「名前修飾」とも呼びます。

バイナリ互換

あるリリースのコンパイラでコンパイルしたオブジェクトファイルを別のリリースのコンパイラを使用してリンクできること。

配列

同じデータ型の値をメモリーに連続して格納するデータ構造。各値にアクセスするには、配列内のそれぞれの値の位置を指定します。

派生クラス

「継承」を参照。

符号化する

「名前の符号化」を参照。

フラグ

「コンパイラオプション」を参照。

プラグマ

コンパイラに特定の処置を指示するコンパイラのプリプロセッサ命令、または特別な注釈。

べき等

ヘッダーファイルの属性。ヘッダーファイルを 1 つの翻訳単位に何回インクルードしても、一度インクルードした場合と同じ効果を持つこと。

変数

識別子で命名されているデータ項目。各変数は `int` や `void` などの型とスコープを持っています。「クラス変数」、「インスタンス変数」、「局所変数」も参照。

マルチスレッド

シングルまたはマルチプロセッサシステムで並列アプリケーションを開発・実行するためのソフトウェア技術。

メソッド

一部のオブジェクト指向言語でメンバー関数の代わりに使用される用語。

メンバー関数

クラスの要素である関数。データ定義や型定義と区別してこのように呼ばれます。

リンカー

オブジェクトコードとライブラリを結び付けて、完全な実行可能プログラムを作成するツール。

例外

プログラムの通常の流れの中で起こる、プログラムの継続を妨げるエラー。たとえば、メモリーの不足やゼロ除算などを指します。

例外処理

エラーの捕捉と防止を行うためのエラー回復処理。具体的には、プログラムの実行中にエラーが検出されると、あらかじめ登録されている例外ハンドラにプログラムの制御が戻り、エラーを含むコードは実行されなくなることを指します。

例外ハンドラ

エラーを処理するために作成されたコード。ハンドラは、対象とする例外が起こると自動的に呼び出されます。

ロケール

地理的な領域と言語のどちらか、あるいはその両方に固有な一連の規約。日付、時刻、通貨単位などの形式。

索引

記号

! NOT 演算子、iostream 200, 206
<< 挿入演算子
  complex 231
  iostream 198, 200
>> 抽出演算子
  complex 231
  iostream 202
#pragma align 423
#pragma end_dumpmacros 427
#pragma fini 427
#pragma ident 428
#pragma init 429
#pragma no_side_effect 429
#pragma pack 430
#pragma rarely_called 432
#pragma returns_new_memory 432
#pragma unknown_control_flow 433
#pragma weak 433
#pragma キーワード 422 - 435
+e(0|1)、コンパイラオプション 258
+p、コンパイラオプション 306
+w2、コンパイラオプション 321
+w、コンパイラオプション 85, 320

数字

-386、コンパイラオプション 247
-486、コンパイラオプション 247

A

__ARRAYNEW、事前定義マクロ 254
-a、コンパイラオプション 247
.a、ファイル名接尾辞 19, 235

B

-B*binding*、コンパイラオプション 107, 248 - 250
bool 型とリテラル、許可する 265
_BOOL、事前定義マクロ 254
__BUILTIN_VA_ARG_INCR、事前定義マクロ 254

C

C++ マニュアルページ、アクセス xxxv
C++ 標準ライブラリ 156 - 158
  RogueWave バージョン 176
  置き換え 169, 174
  構成要素 175 - 192
  マニュアルページ 159, 178 - 192
c++、ファイル名接尾辞 18
C++ マニュアルページ、アクセス 159, 160
C99 サポート 358

C API (アプリケーションプログラミングインタフェース)
C++ 実行時ライブラリに依存しないようにする 240
ライブラリの作成 240
CC pragma 指令 422
CCadmin コマンド 86
CCFLAGS、環境変数 30
CC_tmpl_opt、オプションファイル 95
.cc、ファイル名接尾辞 18
cerr、標準ストリーム 144, 196
c_exception、complex クラス 230
-cg、コンパイラオプション 250
char* 抽出子 203 - 204
char の有符号性 signedness 336
char、の有符号性 signedness 336
char の有符号性の保護 336
cin、標準ストリーム 144, 196
clog、標準ストリーム 144, 196
-compat
C++ ライブラリのリンク、のモード 166
-features オプション、値の制限 265
-library オプション、値の制限 295
コンパイラオプション 250
デフォルトでリンクされるライブラリ、への影響 160
ライブラリ、を使用できるモード 156
complex
エラー処理 230 - 231
演算子 227 - 228
効率 233
互換モード 225
混合算術演算 232 - 233
コンストラクタ 226 - 227
三角関数 229 - 230
数学関数 228 - 230
入力/出力、 231 - 232
標準モードと libCstd 225
ヘッダーファイル 226
マニュアルページ 234
ライブラリ 156 - 158, 164 - 166, 225 - 234
ライブラリ、リンク 226
complex_error
定義 230
メッセージ 228
const_cast 演算子 110
cout、標準ストリーム 144, 196
__cplusplus、事前定義マクロ 59, 250, 254
.cpp、ファイル名接尾辞 18
.cxx、ファイル名接尾辞 18
-c、コンパイラオプション 21, 250
C 標準ヘッダーファイル、置き換え 174
.C、ファイル名接尾辞 18

D

-dalign、コンパイラオプション 256
__DATE__、事前定義マクロ 254
-DDEBUG 94
dec、iostream マニピュレータ 213
definition キーワード、テンプレートオプションファイル 97
delete 配列形式、認識する 268
dlopen()、関数呼び出し 236, 237, 239
dmesg、実際の実メモリー 30
double、complex 型の値 226
-dryrun、コンパイラオプション 25, 257
dynamic_cast 演算子 112
-D、コンパイラオプション 34, 253 - 255
+d、コンパイラオプション 252
-d、コンパイラオプション 255

E

EDOM、errno の設定 231
endl、iostream マニピュレータ 213
ends、iostream マニピュレータ 213
enum
スコープ修飾子、として名前を使用 54
前方宣言 52

不完全な、使用 53
ERANGE、errno の設定 231
errno、定義 230 - 231
-erroff、コンパイラオプション 259
error 関数 200
-errtags、コンパイラオプション 260
-errwarn、コンパイラオプション 261
explicit キーワード、認識する 269
export キーワード、認識する 266
-E、コンパイラオプション 257 - 258

F

-fast、コンパイラオプション 262 - 265
-features、コンパイラオプション 49 - 58, 104, 112, 265 - 270
__FILE__、事前定義マクロ 254
-filt、コンパイラオプション 270
-flags、コンパイラオプション 273
floating point
precision (Intel) 280
float 型挿入子、iostream 出力 198
flush、iostream マニピュレータ 201, 213
-fnonstd、コンパイラオプション 274
-fns、コンパイラオプション 274
Fortran 実行時ライブラリ、リンク 358
-fprecision=*p*、コンパイラオプション 276 - 277
-fround=*r*、コンパイラオプション 277 - 278
-fsimple=*n*、コンパイラオプション 278 - 280
-fstore、コンパイラオプション 280
fstream.h
iostream ヘッダーファイル 197
使用 207
fstream、定義 197, 222
-ftrap、コンパイラオプション 280
__func__、識別子 58

G

-G
オプションの説明 282 - 283
動的ライブラリコマンド 237
-g
オプションの説明 283
によるテンプレートのコンパイル 94
get、char 抽出子 204
get ポインタ、streambuf 218
__global 50, 361
-g0 オプションの説明 285
gprof、C++ ユーティリティ 12

H

-help、コンパイラオプション 286
hex、iostream マニピュレータ 213
__hidden 50, 361
-H、コンパイラオプション 285
-h、コンパイラオプション 285

I

__i386、事前定義マクロ 255
i386、事前定義マクロ 255
IA、定義 26
ifstream、定義 197
.il、ファイル名接尾辞 19
include キーワード、テンプレートオプションファイル 96
include ディレクトリ、テンプレート定義ファイル 95
include ファイル、検索順序 286, 287
-inline、-xinline参照
-instances=*a*、コンパイラオプション 87 - 92, 290
-instlib、コンパイラオプション 291
iomanip.h、iostream ヘッダーファイル 198, 213
iostream 136

iostream
library 163
make の使用 32
stdio 206, 217
エラーの処理 205
エラービット 201
機能の拡張、マルチスレッドでの安全性における注意点 147
構造 196 - 197
互換モード 195
コピー 211
コンストラクタ 197
作成 207 - 211
従来の iostream 157, 162, 299
出力エラー 200 - 201
出力のフラッシュ 201
使用 197
新旧の形式が混在する 299
シングルスレッドアプリケーション 137
ストリームの代入 211
定義 222
定義済みの 196
入力 202
標準 iostream 157, 162, 299
標準モード 195, 198, 299
フォーマット 212
ヘッダーファイル 197
マニピュレータ 212
マニュアルページ 195, 219
マルチスレッドで使用しても安全な新しいインタフェース関数 142 - 143
マルチスレッドで使用しても安全なインタフェースの変更 141
マルチスレッドで使用しても安全な再入可能な関数 136
マルチスレッドでの安全性の制約 137
マルチスレッドにおける新しいクラス階層 141
用語 222
ライブラリ 156, 162 - 166
～への出力 198
iostream.h, iostream ヘッダーファイル 144, 197
ISO C++ 標準
準拠 1
単一定義規則 81, 93
ISO10646 UTF-16 文字列リテラル 417
istream クラス、定義 196
istrstream クラス、定義 197
-I-、コンパイラオプション 287
-I、コンパイラオプション 95, 286
-i、コンパイラオプション 290
.i、ファイル名接尾辞 18

K

.KEEP_STATE、標準ライブラリヘッダーファイルでの使用 32
-keeptmp、コンパイラオプション 293
-KPIC、コンパイラオプション 238, 292
-Kpic、コンパイラオプション 238, 292

L

ldd コマンド 169
LD_LIBRARY_PATH 環境変数 168, 236
lex、C++ ユーティリティ 12
libC
MT 環境、における使用 133
互換モード 195, 198
マルチスレッドで使用しても安全な新しいクラス 141
マルチスレッドでの安全性のコンパイルとリンク 136
ライブラリ 156 - 158
libcomplex、complex参照
libCrun ライブラリ 127, 128, 156, 160, 238
libCstd ライブラリ、 C++ 標準ライブラリ参照
libc ライブラリ 155
libdemangle ライブラリ 156 - 159
libgc ライブラリ 156
libiostream、 iostream 参照
libm
インラインテンプレート 363

オプティマイズされたバージョン 363
ライブラリ 155
-libmieee、コンパイラオプション 294
-libmil、コンパイラオプション 294
-library、コンパイラオプション 161 - 162, 166, 167, 294 - 300
librwtool、Tools.h++ も参照
libthread ライブラリ 155
libw ライブラリ 155
limit、コマンド 29
__LINE__、事前定義マクロ 254
linking
iostream library 163
-lpthread と POSIX 294
-lthread
-xnolib によるよくし 167
代わりに -mt を使用 127, 136
-L、コンパイラオプション 160, 293
-l、コンパイラオプション 34, 155, 160, 293 - 294

M

make コマンド 31 - 32
math.h、complex ヘッダーファイル 234
-mc、コンパイラオプション 300
-migration、コンパイラオプション 300
-misalign、コンパイラオプション 301
-mr、コンパイラオプション 302
-mt オプション
オプションの説明 302
および libthread 136
-mt、コンパイラオプション
ライブラリのリンク 155
mutable キーワード、認識する 266

N

namespace キーワード、認識する 269
-native、コンパイラオプション 303
new 配列形式、認識する 268
nocheck、フラグ 99
-noex、コンパイラオプション 303
-nofstore、コンパイラオプション 303 - 304
-nolibmil、コンパイラオプション 304
-nolib、コンパイラオプション 162, 304
-noqueue、コンパイラオプション 304
-norunpath、コンパイラオプション 162, 304

O

oct、iostream マニピュレータ 213
ofstream クラス 207
-O*level*、コンパイラオプション 305
ompat 250
_OPENMP プリプロセッサトークン 379
ostream クラス、定義 196
ostrstream クラス、定義 197
overflow 関数、streambuf 148
-O、コンパイラオプション 305
-o、コンパイラオプション 305
.o ファイル
オプション接尾辞 19
残す 21

P

PATH 環境変数、設定 xxx
-pentium、コンパイラオプション 307
-pg、コンパイラオプション 307
-PIC、コンパイラオプション 307
-pic、コンパイラオプション 307
POSIX と -lpthread 294
private、オブジェクトスレッド 145
prof、C++ ユーティリティ 12
Programming Language–C++、標準の準拠 1
-pta、コンパイラオプション 307
ptclean コマンド 86
pthread_cancel() 関数 129
-pti、コンパイラオプション 95, 308

-pto、コンパイラオプション 308
-ptr、コンパイラオプション 308
-ptv、コンパイラオプション 309
put ポインタ、streambuf 218
-P、コンパイラオプション 306
-p、コンパイラオプション 306

Q

-Qoption、コンパイラオプション 309
-qoption、コンパイラオプション 310
-Qproduce、コンパイラオプション 310
-qproduce、コンパイラオプション 310
-qp、コンパイラオプション 310

R

-readme、コンパイラオプション 311
readme ファイル 2
reinterpret_cast 演算子 110, 325
resetiosflags、iostream マニピュレータ 213
RogueWave
  C++ 標準ライブラリ 176
    Tools.h++ も参照
RTLD_GLOBAL、環境変数 169
rtti キーワード、認識する 269
-R、コンパイラオプション 162, 311

S

-sbfast、コンパイラオプション 312
sbufpub、マニュアルページ 208
-sb、コンパイラオプション 312
setbase、iostream マニピュレータ 213
setfill、iostream マニピュレータ 213
setioflags、iostream マニピュレータ 213
setprecision、iostream マニピュレータ 213
set_terminate() 関数 129
set_unexpected() 関数 129
setw、iostream マニピュレータ 213
skip フラグ、iostream 205
.so.*n*、ファイル名接尾辞 19
Solaris オペレーティング環境ライブラリ 155
.so、ファイル名接尾辞 19, 236
__sparc、事前定義マクロ 255
__sparcv9、事前定義マクロ 255
sparc、事前定義マクロ 255
special、テンプレートのコンパイルオプション 100 - 101
Standard C++ Class Library Reference 176
static_cast 演算子 112
-staticlib、コンパイラオプション 161, 167, 312 - 315
__STDC__、事前定義マクロ 59, 254
stdio
  iostreams の併用 206
    stdiobuf マニュアルページ 217
stdiostream.h、iostream ヘッダーファイル 198
STL (標準テンプレートライブラリ)、構成要素 175
stream.h、iostream ヘッダーファイル 198
streambuf
  get ポインタ 218
  put ポインタ 218
  新しい関数 142
  キュー形式とファイル形式 218
  公開仮想関数 148
  使用 218
  定義 217, 223
  マニュアルページ 218
  ロック 135
stream_locker
  マニュアルページ 147
  マルチスレッドで使用しても安全なオブジェクトとの同期 141
streampos 210
stream、定義 223

strstream.h、iostream ヘッダーファイル 197
strstream、定義 197, 223
struct、名前のない宣言 54
__SUNPRO_CC 251, 254
__SUNPRO_CC_COMPAT=(4|5)、事前定義マクロ 250, 254
__SUNPRO_CC、事前定義マクロ 254
.SUNWCCh、ファイル名接尾辞 172, 173
SunWS_cache 92
SunWS_config ディレクトリ 95
__sun、事前定義マクロ 254
sun、事前定義マクロ 254
__SVR4、事前定義マクロ 254
swap -s、コマンド 28
__symbolic 50, 361
-S、コンパイラオプション 311
-s、コンパイラオプション 312
.S、ファイル名接尾辞 18
.s、ファイル名接尾辞 18

T

tcov、C++ ユーティリティ 12
-temp=*dir*、コンパイラオプション 315
-template、コンパイラオプション 86, 94, 315 - 316
terminate() 関数 129
__thread 51
thr_exit() 関数 129
thr_keycreate、マニュアルページ 145
-time、コンパイラオプション 317
__TIME__、事前定義マクロ 254
Tools.h++
  コンパイラオプション 166
  従来の iostream と標準 iostream 158
  デバッグライブラリ 156
  標準モードと互換モード 158
  マニュアル 158

U

ulimit、コマンド 28
__\'*uname-s*\'_\'*uname-r*\'、事前定義マクロ 255
unexpected() 関数 129
unix、事前定義マクロ 255
__unix、事前定義マクロ 255
UNIX ツール 12
-unroll=*n*、コンパイラオプション 318
-U、コンパイラオプション 34, 317
U"..." 形式の文字列リテラル 417

V

__VA_ARGS__ 識別子 26
-vdelx、コンパイラオプション 318
-verbose、コンパイラオプション 23, 85, 319 - 320
VIS Software Developers Kit 418
-V、コンパイラオプション 318
-v、コンパイラオプション 25, 318

W

_WCHAR_T、事前定義 UNIX シンボル 255
ws、iostream マニピュレータ 205, 213
-w、コンパイラオプション 321

X

-xalias_level、コンパイラオプション 322
-xarch=*isa*、コンパイラオプション 326 - 332
-xar、コンパイラオプション 89, 236 - 237, 325
-xa、コンパイラオプション 322
-xbuiltin、コンパイラオプション 332
-xcache=*c*、コンパイラオプション 333 - 335
-xcg89、コンパイラオプション 335
-xcg92、コンパイラオプション 335
-xchar、コンパイラオプション 336
-xcheck、コンパイラオプション 337

-xchip=*c*、コンパイラオプション 338 - 340
-xcode=*a*、コンパイラオプション 340 - 341
-xcrossfile、コンパイラオプション 342
-xdumpmacros、コンパイラオプション 344
-xe、コンパイラオプション 349
-xF、コンパイラオプション 349 - 351
-xhelp=flags、コンパイラオプション 351
-xhelp=readme、コンパイラオプション 351
-xhreadvar、コンパイラオプション 414
-xia、コンパイラオプション 351
-xildoff、コンパイラオプション 352
-xildon、コンパイラオプション 352
-xinline、コンパイラオプション 353
-xipo、コンパイラオプション 355
-xjobs、コンパイラオプション 358
-xlang、コンパイラオプション 358
-xldscope、コンパイラオプション 50, 360
-xlibmieee、コンパイラオプション 362
-xlibmil、コンパイラオプション 363
-xlibmopt、コンパイラオプション 363
-xlicinfo、コンパイラオプション 365
-xlic_lib、コンパイラオプション 364
-xlinkopt、コンパイラオプション 365
-xM1、コンパイラオプション 368
-xmemalign、コンパイラオプション 369
-xMerge、コンパイラオプション 368
-Xm、コンパイラオプション 322
-xM、コンパイラオプション 367 - 368
-xnativeconnet、コンパイラオプション 370
-xnolibmil、コンパイラオプション 374
-xnolibmopt、コンパイラオプション 374 - 375
-xnolib、コンパイラオプション 162, 167, 372 - 374
-xO*level*、コンパイラオプション 375 - 379
-xopenmp、コンパイラオプション 379
-xpagesize_heap、コンパイラオプション 381
-xpagesize_stack、コンパイラオプション 382
-xpagesize、コンパイラオプション 380
-xpg、コンパイラオプション 388
-xport64、コンパイラオプション 389
-xprefetch_level、コンパイラオプション 396
-xprefetch、コンパイラオプション 393
-xprofile_ircache、コンパイラオプション 400
-xprofile_pathmap、コンパイラオプション 401
-xprofile、コンパイラオプション 397 - 400
- xregs、コンパイラオプション 402
-xregs、コンパイラオプション 239
-xsafe=mem、コンパイラオプション 404 - 405
-xsbfast、コンパイラオプション 405
-xsb、コンパイラオプション 405
-xspace、コンパイラオプション 406
-xs、コンパイラオプション 404
-xtarget=*t*、コンパイラオプション 406 - 413
-xtime、コンパイラオプション 415
-xtrigraphs、コンパイラオプション 415
-xunroll=*n*、コンパイラオプション 417
-xustr、コンパイラオプション 417
-xvis、コンパイラオプション 418
-xwe、コンパイラオプション 419
X 型挿入子、iostream 198

Y

yacc、C++ ユーティリティ 12

Z

-z *arg*、コンパイラオプション 419

あ

アクセスできるマニュアル xxxii
アセンブラ、コンパイル構成要素 25
アセンブリ言語のテンプレート 418

値
cout への挿入 199
double 226
float 198
flush 201
long 216
マニピュレータ 198, 216
値クラス、使用 122
アプリケーション
マルチスレッドで使用しても安全 133
マルチスレッドで使用しても安全な
iostream オブジェクトの使用 149 - 151
マルチスレッドプログラムのリンク 127, 136

い

依存関係
C++ 実行時ライブラリに依存しないようにする 240
共有ライブラリ 238
インクリメンタルリンクエディタ、コンパイル構成要素 25
インスタンス化
オプション 87 - 92
テンプレート関数 71
テンプレート関数メンバー 71
テンプレートクラス 71
テンプレートクラスの静的データメンバー 72
インスタンスの状態、一致した 94
インスタンスメソッド
静的 90
大域 91
テンプレート 87
半明示的 92
明示的 91
インライン関数
C++、使用に適した状況 120
オプティマイザによる 353
インライン展開、アセンブリ言語テンプレート 25

え

エラー
状態、iostreams 200
ビット 201
マルチスレッドでの安全性のチェック 137
エラー処理
complex 230 - 231
エラーの処理
入力 205 - 206
エラーメッセージ
complex_error 228
コンパイラバージョンの互換性問題 19
リンカー 22, 24
演算子
complex 231
iostream 198, 200, 202 - 203
基本算術 227 - 228
スコープ決定 139
演算ライブラリ、複素数 225 - 234

お

オーバーヘッド、マルチスレッドで使用しても安全なクラスのパフォーマンス 141, 139
オブジェクト
stream_locker 147
一時 119
一時オブジェクト、の寿命 268
共有オブジェクトに対処する方法 145
共有オブジェクトの破棄 148
大域共有 144
破棄する順序 268
ライブラリ内、リンク時 235
オブジェクトスレッド、private 145
オブジェクトファイル
再配置可能 238
リンク順序 34
オプション
アルファベット順のオプションリストも参照
拡張コンパイル 262
言語 38
構文形式 33, 245

コード生成 35
最適化 42
サブプログラムのコンパイル 22 - 23
出力 41, 42
処理順序 17, 34
スレッド 46
説明の見出し 246
ソース 45
デバッグ 36
テンプレート 46, 95
テンプレートのコンパイル 89
認識できない 24
廃止 40, 308
パフォーマンス 42
浮動小数点 38
プリプロセッサ 44
プロファイル 45
ライセンス 40
ライブラリ 160 - 162
ライブラリリンク 39
リファレンス 45
オプティマイザのメモリー不足 29

か

外部
インスタンス 87
リンケージ 88
拡張機能 49 - 58
定義 1
非標準コードを許可する 266
数の共役 225
仮想メモリー、制限 28 - 29
各国語のサポート、アプリケーション開発 13
ガベージコレクション
ライブラリ 159, 166
環境変数
SUN_PROFDATA 398
SUN_PROFDATA_DIR 398
CCFLAGS 30
LD_LIBRARY_PATH 168, 236
RTLD_GLOBAL 169
SUNWS_CACHE_NAME 92
関数
streambuf 公開仮想関数 148
置き換え 52
オプティマイザによるインライン展開 353
静的、クラスフレンドとして 57
宣言指定子 50
動的 (共有) ライブラリ 237
マルチスレッドで使用しても安全な公開関数 135
関数、__func__ における名前 58
関数テンプレート 65 - 71
使用 66
宣言 65
定義 66
テンプレートも参照
関数の並べ替え 349
関数レベルの並べ替え 349

き

キャスト
const と volatile 110
reinterpret_cast 110
static_cast 112
動的 112
void* にキャストする 113
下位にキャストする 113
上位にキャストする 113
キャッシュ
オプティマイザ用 333
ディレクトリ、テンプレート 19
境界整列
デフォルト 431
もっとも厳密な 430
共有オブジェクト 145, 148
共有ライブラリ
C プログラムからのアクセス 241
構築 237, 282
構築、例外のある 107
名前 285
のリンクを使用できなくする 255

例外を含む 238
共用体宣言指定子 51
極座標、複素数 225
局所スコープ規則、使用する/使用しない 266

く

空白
行頭 204
抽出子 205
飛ばす 205, 215
区間演算ライブラリ、リンク 351
クラス
間接的に渡す 123
直接的に渡す 124
クラスインスタンス、名前のない 56
クラス言指定子 51
クラステンプレート 67 - 70
使用 69
静的データメンバー 69
宣言 67
定義 67, 68
テンプレートも参照
パラメータ、デフォルト 73
不完全な 67
メンバー、定義 68
クラスライブラリ、使用 162 - 166

け

警告
C ヘッダーの置き換え 174
移植性がないコード 320
移植性を損なう技術的な違反 321
効率が悪いコード 320
認識できない引数 24
廃止されている構文 321
問題がある ARM 言語構造 267
抑止する 321
言語
C99 サポート 358
オプション 38
各国語のサポート 13
言語が混合したリンク 358
検索
テンプレート定義ファイル 94
検索パス
インクルードファイル、定義 286
ソースオプション 45
定義 95
テンプレートオプション 45
動的ライブラリ 162
標準ヘッダーの実装 172 - 173

こ

公開関数、マルチスレッドで使用しても安全 135
構成マクロ 157
構造体宣言指定子 51
構文
CC コマンド 17
オプション 33, 245
コードオプティマイザ、コンパイル構成要素 25
コード生成
インライン機能とアセンブラ、コンパイル構成要素 25
オプション 35
互換性問題、コンパイラバージョン 19
互換モード
-compatも参照
iostream 195
libC 195, 198
libcomplex 225
Tools.h++ 158
国際化、実装 13
コピー
ストリームオブジェクト 211
ファイル 219
コマンド行
オプション、認識できない 24
認識されるファイル接尾辞 18
混合算術演算、複素数演算ライブラリ 232 - 233
コンストラクタ

complex クラス 226
iostream 197
静的 237
コンパイラ
構成要素の起動順序 25
診断 23 - 24
バージョン、互換性問題 19
コンパイル、メモリー条件 27, 30
コンパイル、アクセス xxxi

さ

サイズ、メモリー 431
最適化
levels 375
数学ライブラリ 363
対象ハードウェア 406
のオプション 42
リンク時 365
先読み命令、有効にする 393
サブプログラム、コンパイルオプション 22 - 23
三角関数、複素数演算ライブラリ 229 - 230
3 文字表記シーケンス、認識する 415

し

シーケンス、マルチスレッドで使用しても安全な
入出力操作の実行 144
シェル、仮想メモリーの制限 28
シグナルハンドラ
およびマルチスレッド 128
および例外 103
事前定義マクロ 254
実行可能な接頭辞 385
実行時エラーメッセージ 104
実部、複素数 225, 228
実メモリー、表示 30
シフト演算子、iostreams 214
従来のリンクエディタ、コンパイル構成要素 25
終了関数 427
出力 195
cout 198
エラーの処理 200
オプション 41
出力のフラッシュ 201
バイナリ 202
バッファのフラッシュ 201
初期化関数 429
書体と記号について xxviii
処理順序、オプション 17
シンボル宣言指定子 50
シンボルテーブル、実行可能ファイル 312
シンボル、マクロ参照

す

数学関数、複素数演算ライブラリ 228 - 230
数学ライブラリ、オプティマイズされたバージョン 363
数、複素数 225 - 228
スコープ決定演算子、unsafe_ クラス 139
スタック
のページサイズを設定する 380
スレッドオプション 46
スワップ領域 28 - 29

せ

静的
オブジェクト、非ローカルの初期設定子 267
関数、参照 81
変数、参照 81
静的 (アーカイブ) ライブラリ 235
静的インスタンス 87 - 90
静的データ、マルチスレッドアプリケーション内の 143 - 144
静的テンプレートクラスメンバー 101
静的リンク
コンパイラ提供ライブラリ 161, 312, 315
デフォルトライブラリ 166
テンプレートインスタンス 90

ライブラリの束縛 248
制約、マルチスレッドで使用しても安全な iostream 137
絶対値、複素数 225
接尾辞
.SUNWCCh 172 - 173
makefile 31 - 32
コマンド行ファイル名 18
テンプレート定義ファイル 96
なしのファイル 172
ライブラリ 236
宣言指定子
__global 50, 361
__hidden 50, 361
__symbolic 50, 361
__thread 51

そ

相互排他領域、定義 147
相互排他ロック、マルチスレッドで使用しても安全なクラス 140, 147
挿入
演算子 198 - 199
定義 222
ソースコンパイラオプション 45
ソースファイル
位置規約 94
位置の定義 97 - 99
テンプレート定義 96
リンク順序 34

た

大域
インスタンス 88 - 91
データ、マルチスレッドアプリケーション内の 143 - 144
マルチスレッドで使用しても安全なアプリケーション内の共有オブジェクト 144
リンケージ 88 - 92
代入、iostream 211

ち

中間言語トランスレータ、コンパイル構成要素 25
抽出
char* 203 - 204
演算子 202
空白 205
定義 222
ユーザー定義の iostream 202 - 203

て

定義済みマニピュレータ、iomanip.h 213
定義、テンプレートの検索 94
データ型、複素数 225
適用子、引数付きマニピュレータ 216
デストラクタ、静的 237
デバッグ
オブジェクトファイルなし 404
オプション 36
プログラムの準備 23, 284
デフォルト演算子、使用 121
デフォルトライブラリ、静的リンク 166
テンプレート
入れ子 72
インスタンスメソッド 87, 94
インライン展開 363
オプション 46
オプションファイルの共有 96
キャッシュディレクトリ 19
コマンド 86
コンパイル 89
冗長コンパイル 85
静的オブジェクト、参照 81
ソースファイル 94, 97 - 99
定義分離型と定義取り込み型の編成 94
特殊エントリ 99 - 101
特殊化 73
標準テンプレートライブラリ (STL) 175
部分特殊化 74
リンク 23

レポジトリ 92
テンプレート定義
検索パス 95
含める 62
別の、ファイル 94
テンプレートのインスタンス化 70
暗黙の 70
関数 70
全クラス 86
明示的 70
テンプレートのプリリンカー、コンパイル構成要素 25
テンプレートの問題 75
静的オブジェクト、参照 81
テンプレート関数のフレンド宣言 77
テンプレート定義内での修飾名の使用 80
引数としての局所型 77
非局所型名前の解決とインスタンス化 75

と

動的 (共有) ライブラリ 167, 237, 248, 285
トークンのスペル、代替 265
トラップモード 280

な

内部手続きアナライザ 25
内部手続きオプティマイザ 355
名前のないクラスインスタンス、受け渡し 56

に

入出力ライブラリ 195
入力
`iostream` 202
エラーの処理 205 - 206
先読み 205
バイナリ 204
入力/出力、`complex` 195, 231 - 232
入力データの先読み 205

ね

ネイティブコネクタツール (NCT) 370

は

は 336
ハードウェアのアーキテクチャ 406
廃止されている構文、許可しない 265
配置、テンプレートインスタンス 87
バイナリ入力、読み込み 204
バッファ
出力のフラッシュ 201
定義 222
パフォーマンス
オプション 42
マルチスレッドで使用しても安全なクラスのオーバーヘッド 139, 141
半明示的インスタンス 88, 92

ひ

ヒープ、のページサイズを設定する 380
引数付きのマニピュレータ、`iostreams` 215 - 217
引数なしのマニピュレータ、`iostreams` 214 - 215
左シフト演算子
`complex` 231
`iostream<Default Para Font` 198
非標準機能 49 - 58
定義 1
非標準コードを許可する 266
標準 C++ ライブラリ・ユーザーズガイド 176
標準 `iostream` クラス 195
標準エラー、`iostreams` 196
標準出力、`iostreams` 196
標準、準拠 1

標準ストリーム、iostream.h 144
標準テンプレートライブラリ (STL) 175
標準入力、iostreams 196
標準ヘッダー
　置き換え 173
　実装 171
標準モード
　-compatも参照
　iostream 195, 198
　libCstd 225
　Tools.h++ 158

ふ

ファイル
　C 標準ヘッダーファイル 172
　fstreamsの使用 207
　位置の再設定 210
　オープンとクローズ 210
　オブジェクト 21, 34, 238
　コピー 208, 219
　実行可能プログラム 21
　ソースファイルも参照
　テンプレートオプション 95
　標準ライブラリ 172
　複数のソース、使用 19
ファイル記述子、使用 210
ファイル内の位置の再設定、fstream 210
ファイル名
　.SUNWCCh、ファイル名接尾辞 172 - 173
　接尾辞 18
　テンプレート定義ファイル 94
フォーマットの制御、iostreams 212
複数のソースファイル、使用 19
複素数データ型 225
浮動小数点
　オプション 38
　不正 280
プリコンパイルヘッダーファイル 383
プリプロセッサ
　オプション 44
　に対してマクロを定義する 253
プログラム
　基本的構築手順 15 - 17
　マルチスレッドプログラムの構築 127
プロセッサ、対象システムを指定する 406
プロファイルオプション 45, 397
フロントエンド、コンパイル構成要素 25

へ

ページサイズ、スタックとヒープ用に設定する 380
べき等 59
ヘッダーファイル
　complex 234
　C 標準 172
　iostream 144, 197, 198, 213
　言語に対応した 59
　作成 59
　標準ライブラリ 170, 176 - 177
　べき等 61
別名、によるコマンドの簡略化 30
偏角、複素数 225
変更可能な引数のリスト 26
変数、スレッドローカル記憶装置指定子 51
変数宣言指定子 50
変数のスレッドローカル記憶装置 51

ま

マクロ
　アルファベット順のマクロリストも参照
　事前定義 254
マニュアル索引 xxxii
マニピュレータ
　iostream 212 - 217
　定義済みの 213
　引数なしの 214
マニュアル、アクセス xxxii
マニュアルページ
　C++ 標準ライブラリ 178 - 192
　complex 234

iostream 195, 208, 212, 216
アクセス 2, 159
マニュアルページ、アクセス xxx
マニュアルへのアクセス xxxiv
マルチスレッド
アプリケーション 128
コンパイル 128
例外処理 129
マルチスレッドで使用しても安全
クラス、派生における注意事項 147
アプリケーション 133
オブジェクト 133
公開関数 135
パフォーマンスオーバーヘッド 139, 141
ライブラリ 133
マルチメディアタイプ、の処理 418

み
右シフト演算子
complex 231
iostream 202

め
明示的インスタンス 88 - 91
メモリーサイズ 431
メモリー条件 27, 29
メンバー変数、キャッシュ 124

も
文字、単一読み込み 204

ゆ
ユーザー定義型
iostream 198
マルチスレッドで使用しても安全 138 - 139
優先順位、の問題回避 199

ら
ライセンス
オプション 40
情報 365
必要条件 3
ライブラリ
C++ コンパイラ、に添付 156
C++ 標準 175 - 192
C インタフェース 155
-mt とのリンク 155
Sun Performance Library、リンク 295, 364
置き換え、C++ 標準ライブラリの置き換え 169 - 174
共有 167 - 169, 255
共有ライブラリの名前 285
区間演算 351
クラス、使用 162
構成マクロ 157
最適化された数学ルーチン 363
従来の iostream 195 - 223
使用 155 - 169
静的 248
接尾辞 235
動的にリンクされる 169
とは 235 - 236
リンクオプション 39, 166
リンク順序 34
ライブラリ、構築
C API を持つ 240
公開 239
静的 (アーカイブ) 235 - 237
動的 (共有) 235 - 238
非公開 239
リンクオプション 283
例外を含む共有 238

り
リテラル文字列を読み取り専用メモリーに 265
リファレンスオプション 45
リンク
complex ライブラリ 164, 166

-mt オプション 136
高速化 404
コンパイルからの分離 21
コンパイルとの整合性 22 - 23
システムライブラリを使用しない 372
シンボリック 172
静的 (アーカイブ) ライブラリ 161, 166, 235, 248, 312, 315
テンプレートインスタンスメソッド 87
動的 (共有) ライブラリ 169, 236, 248
マルチスレッドで使用しても安全な libC ライブラリ 136
ライブラリ 155, 160, 166
ライブラリオプション 39
リンク時の最適化 365

れ

例外
longjmp および 106
setjmp および 106
およびマルチスレッド 129
関数、置き換えでの 52
共有ライブラリ 238
許可しない 266
シグナルハンドラおよび 106
定義済みの 105
トラップ 280
のある共有ライブラリの構築 107
標準クラス 105
標準ヘッダー 105
無効化 104

ろ

ロック
streambuf 135
stream_locker も参照
オブジェクト 145 - 147
相互排他 140, 147

わ

ワークステーション、メモリー条件 29

.c、ファイル名接尾辞 18
従来の 299
表に 142