C++ ユーザーズガイド

Forte Developer 7

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

Part No. 816-4919-10
2002 年 6 月 Revision A

原典： *C++ User's Guide*
Part No: 806-2460-10
Revision A

目次

はじめに　xxv

内容の紹介　xxv

書体と記号について　xxvi

シェルプロンプトについて　xxviii

Forte Developer の開発ツールとマニュアルページへのアクセス　xxviii

Forte Developer マニュアルへのアクセス　xxx

関連する Solaris マニュアル　xxxiii

市販の書籍　xxxiv

ご意見の送付先　xxxv

Part I　Sun WorkShop 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

C++ ユーティリティ　6

各国語のサポート　6

2. C++ コンパイラの使用方法　7

コンパイル方法の概要　7

コンパイラの起動　9

コマンド構文　9

ファイル名に関する規則　10

複数のソースファイルの使用　11

バージョンが異なるコンパイラでのコンパイル　11

キャッシュの衝突　12

コンパイルとリンク　13

コンパイルとリンクの流れ　13

コンパイルとリンクの分離　14

コンパイルとリンクの整合性　14

SPARC V9 のためのコンパイル　15

コンパイラの構成　16

指示および名前の前処理　20

プラグマ　20

`#define` の変数引数リスト　20

事前に定義されている名前　20

#error　21

メモリー条件　21

スワップ領域のサイズ　21

スワップ領域の増加　22

仮想メモリーの制御　22

メモリー条件　23

コマンドの簡略化　23

C シェルでの別名の使用　24

CCFLAGS によるコンパイルオプションの指定　24
make の使用　25

3. C++ コンパイラオプションの使い方　27

構文　27
一般的な注意事項　28
機能別に見たオプションの要約　28
コード生成オプション　29
デバッグオプション　30
浮動小数点オプション　31
言語オプション　31
ライブラリオプション　32
ライセンスオプション　33
廃止オプション　34
出力オプション　34
パフォーマンスオプション　35
プリプロセッサオプション　37
プロファイルオプション　37
リファレンスオプション　38
ソースオプション　38
テンプレートオプション　38
スレッドオプション　39

Part II　C++ プログラムの作成

4. 言語拡張　43

例外の制限の少ない仮想関数による置き換え　43
enum 型と enum 変数の前方宣言　44
不完全な enum 型の使用　45

enum 名のスコープ修飾子としての使用　45
名前のない struct 宣言の使用　46
名前のないクラスインスタンスのアドレスの受け渡し　47
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　48
事前定義済み __func__ シンボルの関数名としての使用　49

5. プログラムの編成　51

ヘッダーファイル　51
言語に対応したヘッダーファイル　51
べき等ヘッダーファイル　53
テンプレート定義　53
テンプレート定義の取り込み　54
テンプレート定義の分離　55

6. テンプレートの作成と使用　57

関数テンプレート　57
関数テンプレートの宣言　57
関数テンプレートの定義　58
関数テンプレートの使用　58
クラステンプレート　59
クラステンプレートの宣言　59
クラステンプレートの定義　59
クラステンプレートメンバーの定義　60
クラステンプレートの使用　61
テンプレートのインスタンス化　62
テンプレートの暗黙的インタンス化　62
全クラスインスタンス化　62
テンプレートの明示的インスタンス化　63

テンプレートの編成　64
デフォルトのテンプレートパラメータ　65
テンプレートの特殊化　65
テンプレートの特殊化宣言　66
テンプレートの特殊化定義　66
テンプレートの特殊化の使用とインスタンス化　67
部分特殊化　67
テンプレートの問題　68
非局所型名前の解決とインスタンス化　68
テンプレート引数としての局所型　70
テンプレート関数のフレンド宣言　70
テンプレート定義内での修飾名の使用　73
テンプレート宣言の入れ子　73
静的変数や静的関数の参照　74
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　74

7. テンプレートのコンパイル　77

冗長コンパイル　77
テンプレートコマンド　78
テンプレートインスタンスの配置とリンケージ　78
外部インスタンスリンケージ　79
静的インスタンス　79
大域インスタンス　80
明示的インスタンス　80
半明示的インスタンス　81
テンプレートレポジトリ　82
レポジトリの構造　82
テンプレートレポジトリへの書き込み　82

複数のテンプレートレポジトリからの読み取り　82
テンプレートレポジトリの共有　83
テンプレート定義の検索　83
ソースファイルの位置規約　83
定義検索パス　84
テンプレートインスタンスの自動一貫性　84
コンパイル時のインスタンス化　84
テンプレートオプションファイル　85
コメント　86
インクルード　86
ソースファイルの拡張子　86
定義ソースの位置　87
テンプレートの特殊化エントリ　90

8. 例外処理　93

同期例外と非同期例外　93
実行時エラーの指定　93
例外の無効化　94
実行時関数と事前定義済み例外の使用　95
シグナルや `Setjmp/Longjmp` と例外との併用　96
例外のある共有ライブラリの構築　97

9. キャスト演算　99

`const` キャスト　100
解釈を変更するキャスト　100
静的キャスト　102
動的キャスト　102
階層の上位にキャストする　103

void* にキャストする　103
階層の下位または全体にキャストする　103

10. プログラムパフォーマンスの改善　109

一時オブジェクトの回避　109
インライン関数の使用　110
デフォルト演算子の使用　111
値クラスの使用　112
クラスを直接渡す　113
各種のプロセッサでクラスを直接渡す　114
メンバー変数のキャッシュ　114

11. マルチスレッドプログラムの構築　117

マルチスレッドプログラムの構築　117
マルチスレッドコンパイルの確認　118
C++ サポートライブラリの使用　118
マルチスレッドプログラムでの例外の使用　119
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　119
マルチスレッド環境での従来の iostream の使用　123
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　123
iostream ライブラリのインタフェースの変更　131
大域データと静的データ　134
連続実行　135
オブジェクトのロック　135
マルチスレッドで使用しても安全なクラス　137
オブジェクトの破棄　138
アプリケーションの例　139

Part III　ライブラリ

12. ライブラリの使用　145

C ライブラリ　145

C++ コンパイラ付属のライブラリ　146

C++ライブラリの説明　146

C++ ライブラリのマニュアルページへのアクセス　148

デフォルトの C++ ライブラリ　149

関連するライブラリオプション　149

クラスライブラリの使用　151

`iostream` ライブラリ　151

`complex` ライブラリ　153

C++ライブラリのリンク　154

標準ライブラリの静的リンク　155

共有ライブラリの使用　156

C++ 標準ライブラリの置き換え　158

置き換え可能な対象　158

置き換え不可能な対象　158

代替ライブラリのインストール　159

代替ライブラリの使用　159

標準ヘッダーの実装　160

13. C++ 標準ライブラリの使用　165

C++ 標準ライブラリのヘッダーファイル　166

C++ 標準ライブラリのマニュアルページ　168

STLport　183

14. 従来の `iostream` ライブラリの使用　185

共有版の `libiostream`　186

定義済みの `iostream`　187

iostream 操作の基本構造　187
従来型の iostream ライブラリの使用　188
iostream を使用した出力　189
iostream を使用した入力　193
ユーザー定義の抽出演算子　194
char* の抽出子　195
1 文字の読み込み　195
バイナリ入力　196
入力データの先読み　196
空白の抽出　196
入力エラーの処理　197
iostream と stdio の併用　197
iostream の作成　198
クラス fstream を使用したファイル操作　198
iostream の代入　202
フォーマットの制御　203
マニピュレータ　203
引数なしのマニピュレータの使用法　205
引数付きのマニピュレータの使用法　206
strstream： 配列用の iostream　208
stdiobuf: 標準入出力ファイル 用の iostream　208
streambuf　208
streambuf の機能　209
streambuf の使用　209
iostream に関するマニュアルページ　210
iostream の用語　213

15. 複素数演算ライブラリの使用　215

複素数ライブラリ　215
複素数ライブラリの使用方法　216
complex 型　216
complex クラスのコンストラクタ　216
算術演算子　217
数学関数　218
エラー処理　220
入出力　221
混合演算　222
効率　223
複素数のマニュアルページ　224
16. ライブラリの構築　225
ライブラリとは　225
静的 (アーカイブ) ライブラリの構築　226
動的 (共有) ライブラリの構築　227
例外を含む共有ライブラリの構築　228
非公開ライブラリの構築　229
公開ライブラリの構築　229
C API を持つライブラリの構築　230
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　231
Part IV　付録
A. C++ コンパイラオプション　235
オプション情報の構成　236
オプションの一覧　237
-386　237
-486　237

-a 237

-B*binding* 238

-c 240

-cg{89|92} 240

-compat [={4|5}] 240

+d 242

-D[]*name*[=*def*] 243

-d{y|n} 245

-dalign 246

-dryrun 247

-E 247

+e{0|1} 248

-fast 249

-features=*a*[,*a*...] 252

-filt[=*filter*[,*filter*...]] 257

-flags 260

-fnonstd 260

-fns[={no|yes}] 260

-fprecision=*p* 262

-fround=*r* 263

-fsimple[=*n*] 264

-fstore 266

-ftrap=*t*[, *t*...] 266

-G 268

-g 269

-g0 271

-H 271

-h[ ]*name* 271

-help 272

-I*pathname* 272

-I- 273

-i 276

-inline 276

-instances=*a* 276

-keeptmp 277

-KPIC 277

-Kpic 277

-L*path* 277

-l*lib* 278

-libmieee 279

-libmil 279

-library=*l*[, *l*...] 279

-mc 284

-migration 285

-misalign 285

-mr[, *string*] 286

-mt 286

-native 287

-noex 287

-nofstore 287

-nolib 288

-nolibmil 288

-noqueue 288

-norunpath 288

-O 289

-O*level* 289

-o*filename* 289

+p 290

-P 290

-p 291

-pentium 291

-pg 291

-PIC 291

-pic 291

-pta 292

-pti*path* 292

-pto 292

-ptr 292

-ptv 293

-Qoption *phase option*[,*option*...] 293

-qoption *phase option* 294

-qp 294

-Qproduce *sourcetype* 294

-qproduce *sourcetype* 295

-R*pathname*[*: pathname...*] 295

-readme 295

-S 296

-s 296

-sb 296

-sbfast 296

-staticlib=*l*[, *l*...] 296

-temp=*path* 299

-template=*opt*[,*opt*...] 299

-time 300

-U*name* 300

-unroll=*n* 301

-V 301

-v 301

-vdelx 301

-verbose=*v*[, *v*...] 302

+w 303

+w2 303

-w 304

-xa 304

-xalias_level[=*n*] 305

-xar 307

-xarch=*isa* 308

-xbuiltin[={%all|%none}] 313

-xcache=*c* 314

-xcg89 316

-xcg92 316

-xcheck[=*i*] 317

-xchip=*c* 317

-xcode=*a* 319

-xcrossfile[=*n*] 321

-xF 322

-xhelp=flags 322

-xhelp=readme 323

-xildoff 324

-xildon 324

-xinline[=*func_spec*[,*func_spec*...]] 325

-xipo[={0|1}] 327

-xlang=*language*[, *language*] 329

-xlibmieee 330

-xlibmil 331

-xlibmopt 331

-xlic_lib=sunperf 332

-xlicinfo 333

-xM 333

-xM1 333

-xMerge 333

-xnativeconnect[=*i*] 334

-xnolib 335

-xnolibmil 338

-xnolibmopt 338

-xopenmp[=*i*] 338

-x*Olevel* 339

-xpg 343

-xprefetch[=*a*[,*a*]] 343

-xprefetch_level[=*i*] 346

-xprofile=*p* 347

-xregs=*r*[, *r*...] 350

-xs 352

-xsafe=mem 352

-xsb 353

-xsbfast 353

-xspace 353

-xtarget=*t* 353

-xtime 360

-xunroll=*n* 361

-xtrigraphs[=(yes|no)] 361

-xvector[=(yes|no)] 362

-xwe 363

-z[ ]*arg* 363

-ztext 363

B. プラグマ 365

プラグマの書式 365

プラグマの詳細 366

#pragma align 366

#pragma init 367

#pragma fini 368

#pragma ident 368

#pragma no_side_effect 368

#pragma pack(*n*) 369

#pragma returns_new_memory 371

#pragma unknown_control_flow 372

#pragma weak 372

#pragma weak *name* 373

#pragma weak *name1* = *name2* 373

用語集 375

索引 385

表目次

表 P-1　書体表記　xxvi
表 P-2　コード表記　xxvii
表 P-3　C++ 関連のマニュアルページ　xxxiii
表 2-1　C++ コンパイラが認識できるファイル名接尾辞　10
表 2-2　C++ コンパイルシステムの構成要素　19
表 3-1　オプションの構文形式の例　27
表 3-2　コード生成オプション　29
表 3-3　デバッグオプション　30
表 3-4　浮動小数点オプション　31
表 3-5　言語オプション　31
表 3-6　ライブラリオプション　32
表 3-7　ライセンスオプション　33
表 3-8　廃止オプション　34
表 3-9　出力オプション　34
表 3-10　パフォーマンスオプション　35
表 3-11　プリプロセッサオプション　37
表 3-12　プロファイルオプション　37
表 3-13　リファレンスオプション　38
表 3-14　ソースオプション　38
表 3-15　テンプレートオプション　38
表 3-16　スレッドオプション　39

表 10-1　アーキテクチャ別の構造体と共用体の渡し方　114
表 11-1　iostream の中核クラス　124
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　125
表 12-1　C++ コンパイラに添付されるライブラリ　146
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　154
表 12-3　ヘッダー検索の例　161
表 13-1　C++ 標準ライブラリのヘッダーファイル　166
表 13-2　C++ 標準ライブラリのマニュアルページ　168
表 14-1　iostream ルーチンのヘッダーファイル　188
表 14-2　iostream の定義済みマニピュレータ　204
表 14-3　iostream に関するマニュアルページの概要　211
表 14-4　iostream の用語　213
表 15-1　複素数ライブラリの関数　219
表 15-2　複素数の数学関数と三角関数　219
表 15-3　複素数ライブラリ関数　221
表 15-4　complex 型のマニュアルページ　224
表 A-1　オプション構文形式の例　235
表 A-2　オプションの見出し　236
表 A-3　SPARC と IA 用の事前定義シンボル　244
表 A-4　-fast 展開　249
表 A-5　互換モードと標準モードでの -feature オプション　252
表 A-6　標準モードだけに使用できる -features オプション　255
表 A-7　互換モードだけに使用できる -features オプション　255
表 A-8　filt オプション　258
表 A-9　互換モードでの -library オプション　279
表 A-10　標準モードでの -library オプション　280
表 A-11　SPARC プラットフォームでの -xarch の値　309
表 A-12　IA プラットフォームでの -xarch 値　312
表 A-13　-xcheck の値　317
表 A-14　-xchip オプション　318
表 A-15　-xcode オプション　320

表 A-16　-xinline オプション　325
表 A-17　-xprefetch の値　343
表 A-18　-xprefecth_level の値　347
表 A-19　-xprofile オプション　348
表 A-20　SPARC プラットフォームの -xtarget の値　354
表 A-21　-xtarget の SPARC プラットフォーム名　355
表 A-22　IA プラットフォームの -xtargetの値　359
表 A-23　Intel アーキテクチャでの -xtarget の展開　359
表 B-1　プラットフォームの最も厳密な境界整列　370
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)　370

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　70

コード例 6–2　フレンド宣言の問題の例　71

コード例 7-1　冗長な `definition` エントリ　87

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　88

コード例 7-3　テンプレートメンバー関数の定義　88

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　89

コード例 7-5　`nocheck` オプション　90

コード例 7-6　`special` エントリ　90

コード例 7-7　`special` エントリを使用する必要がある場合　91

コード例 7-8　`special` エントリの多重定義　91

コード例 7-9　テンプレートクラスの特殊化　92

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　92

コード例 11–1　エラー状態のチェック　127

コード例 11–2　`gcount` の呼び出し　128

コード例 11–3　ユーザー定義の入出力操作　128

コード例 11–4　マルチスレッドでの安全性の無効化　130

コード例 11–5　「マルチスレッドで使用すると安全ではない」への切り換え　130

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　131

コード例 11–7　新しいクラス　131

コード例 11–8　新しいクラス階層　132

コード例 11–9　新しい関数　132

コード例 11–10　ロック処理の使用例　136
コード例 11–11　入出力操作とエラーチェックの不可分化　137
コード例 11–12　共有オブジェクトの破棄　138
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　139
コード例 14-1　string の抽出演算子　194
コード例 A-1　プルプロセッサのプログラム例 foo.cc　247
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　248

はじめに

このマニュアルでは、Forte Developer 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。

Sun WorkShop 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、複合演算ライブラリ (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。

プラグマ: 付録 B ではプラグマに関する情報を記載します。

用語集: 用語集では、C++ およびこのマニュアルで使用される関連用語を説明します。

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。

表 P-1　書体表記

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』

表 P-1　書体表記

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」

表 P-2　コード表記

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)	#

Forte Developer の開発ツールとマニュアルページへのアクセス

Forte Developer の製品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。Forte Developer のコンパイラとツールにアクセスするには、PATH 環境変数に Forte Developer コンポーネントディレクトリを必要とします。Forte Developer マニュアルページにアクセスするには、PATH 環境変数に Forte Developer 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 Forte Developer 製品が /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。 Forte Developer 製品が /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。

Forte Developer 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 Forte Developer 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 Forte Developer 開発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 Forte Developer の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

Forte Developer マニュアルページへのアクセス方法

Forte Developer 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 Forte Developer 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
   ```

Forte Developer マニュアルへのアクセス

Forte Developer の製品マニュアルには、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。

アクセスできる製品マニュアル

Forte Developer 7 製品マニュアルは、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。アクセス可能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	アクセス可能な形式と格納場所
マニュアル (サードパーティ製マニュアルは除く)	形式：HTML 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 インストール製品について 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 インストール製品について 場所： `file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

Forte Developer の関連マニュアル

下表に、マニュアル索引 (`file:/opt/SUNWspro/docs/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。表 P-3 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。

表 P-3　C++ 関連のマニュアルページ

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
`gprof`	プログラムの実行プロファイルの作成

表 P-3　C++ 関連のマニュアルページ

タイトル	内容
`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』トッパン、Margaret A. Ellis、Bjarne Stroustrup 共著、1990 年

『 C++ Programming Language 』第 3 版 Bjarne Stroustrup著、Addison-Wesley、1997年

『C++ プライマー』第 3 版、トッパン、Stanley B. Lippman、Josee Lajoie 共著、1998 年

『Effective C++—50 Ways to Improve Your Programs and Designs』Second Edition、Scott Meyers 著、Addison-Wesley、1998 年

『The C++ Standard Library』Nicolai Josuttis 著、Addison-Wesley、1999 年

『Generic Programming and the STL』Matthew Austern 著、Addison-Wesley、1999 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

『Design Patterns: Elements of Reusable Object-Oriented Software』Erich Gamma、Richard Helm、Ralph Johnson,、John Vlissides 共著、Addison-Wesley、1998 年

『More Effective C++ - 35 Ways of Improve Your Programs and Designs』Scott Meyers 著、Addison-Wesley、1996 年

『Efficient C++: Performance Programming Techniques』Dov Bulka and David Mayhew 共著、Addison-Wesley、2000 年

ご意見の送付先

米国 Sun Microsystems, Inc. では、マニュアルの向上に力を注いでおり、ユーザーのご意見やご提案をお待ちしております。ご意見などがありましたら、次のアドレスまで電子メールをお送りください。

`docfeedback@sun.com`

PART I

Sun WorkShop 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。

- C++ での OpenMP サポート (SPARC)

  このリリースの C++ コンパイラは、明示的な並列化用の OpenMP インタフェースを実装しています。ソースコード指令、実行時ライブラリルーチン、環境変数を次のオプションで指定します。

  ```
  CC -xopenmp[=i]
  ```

詳細は、 338 ページの「-xopenmp[=i]」 を参照してください。

- 型に基づく別名の解析および最適化 (SPARC)

  C++ コンパイラで、-xnoalias オプションを使用できるようになりました。次のようにこのオプションを指定すると、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。

  ```
  CC -xnoalias[=i]
  ```

  詳細は、305 ページの「-xalias_level[=n]」 を参照してください。

- Native Connector Tool のサポート

  新しい -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とそれ以降の共有ライブラリを取り込み、共有ライブラリを Java[tm] 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。-xnativeconnect オプションを指定した場合は、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。Native Connector Tool (NCT) を使用して、C++ 標準ライブラリを Java コードから呼び出すことができるように、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 の使用方法の詳細については、Forte Developer のオンラインヘルプを参照してください。

  詳細は、 334 ページの「-xnativeconnect[=i]」 を参照してください。

- 内部手続きの最適化の拡張 (SPARC)

  -xipo オプションでの最適化レベルが拡張され、すべてのソースファイルでのインライン化をコンパイラで実行できるようになりました。-xipo=2 を指定した場合は、コンパイラは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。

  詳細は、 327 ページの「-xipo[={0|1}]」 を参照してください。

- より詳細な先読み設定

  新しい -xprefetch_level=n 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。n には 1、2、3 のいずれかを指定します。-xprefetch_level が高くなるほど、コンパイラはより多くの先読みを挿入します。

  詳細は、 346 ページの「-xprefetch_level[=i]」 を参照してください。

- スタックオーバーフローのチェック

  新しい -xcheck=stkovf オプション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。

  詳細は、317 ページの「-xcheck[=i]」 を参照してください。

- STLport 標準ライブラリのサポート

  C++ コンパイラで、STLport の標準ライブラリのバージョン 4.5.2 がサポートされました。libCstd が旧バージョン同様デフォルトのライブラリですが、選択的に STLport も使用できるようになりました。このリリースでは、libstlport.a という静的アーカイブと libstlport.so という動的ライブラリの両方が含まれています。

  詳細については、279 ページの「-library=l[, l...]」 を参照してください。

- +w オプションの適用範囲の拡大

  +w オプションでは、関数が大きすぎてインライン化できない場合、およびパラメタが未使用の場合のレポートが生成されなくなりました。これは、出力メッセージを削減し、ルーチン構築で +w オプションを簡単に使用できるようにするためです。

- +w2 オプションの単純化による適用範囲の拡大

  +w2 オプションでは、ルーチン構築で +w2 オプションを簡単に使用できるように、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。

- 改善された #error 指令によるコンパイルの即時中止

  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、障害をレポートします。

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。*source-files* にはソースファイル、*object-files* にはオブジェクトファイル、*libraries* にはライブラリを指定します。

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。

- すべてのリンカーのオプション、-features、-I、-l、-L、-library、-pti、-R、-staticlib、-U、-verbose および -xprefetch は上書きされずに累積される
- -U オプションは、すべて -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++	プリプロセッサの出力ファイルを C++ ソースファイルとして扱い、.c 接尾辞と同じ処理をする。
.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。

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、キャッシュのバージョンに問題があれば、エラーメッセージを発行します。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、C++ コンパイラ 5.3 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 5.3 が 5.0 のキャッシュディレクトリを、C++ コンパイラ 5.3 が使用できるディレクトリ形式に変換するためです。

C++ コンパイラ 5.0 は、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ときは、テンプレートキャッシュディレクトリが格納されているディレクトリごとに `CCadmin -clean` を実行する習慣を付けることをお勧めします (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)。`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。

キャッシュの衝突

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。デフォルトの `-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。

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。

オブジェクトファイル (file1.o、file2.o、file3.o) はコンパイルの後も消去されないので、ファイルを簡単に再リンクしたり、再コンパイルすることができます。

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
- -fast
- -g
- -g0
- -library
- -misalign
- -mt
- -p
- -xa
- -xarch
- -xcg92 および -xcg89
- -xipo
- -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もモジュールも、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 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 オプションまたは、-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。
- オプション以外のもの (先頭にダッシュ (–) もプラス符号 (+) も付いていないもの) には警告は生成されません (ただし、それらはリンカーに渡されます。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 と入力するつもりだった
move.C:
ld: 重大なエラー : ファイル fast: ファイルをオープンできません :
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。CC コマンドは、これらの構成要素をそれぞれ自動的に起動します (コマンド行オプショ

ンを使用して自動起動されないように指定することもできます)。図 2-1 に C++ コンパイルの流れを示します。図 2-1 に、構成要素がコンパイラから呼び出される順番を示します。

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。

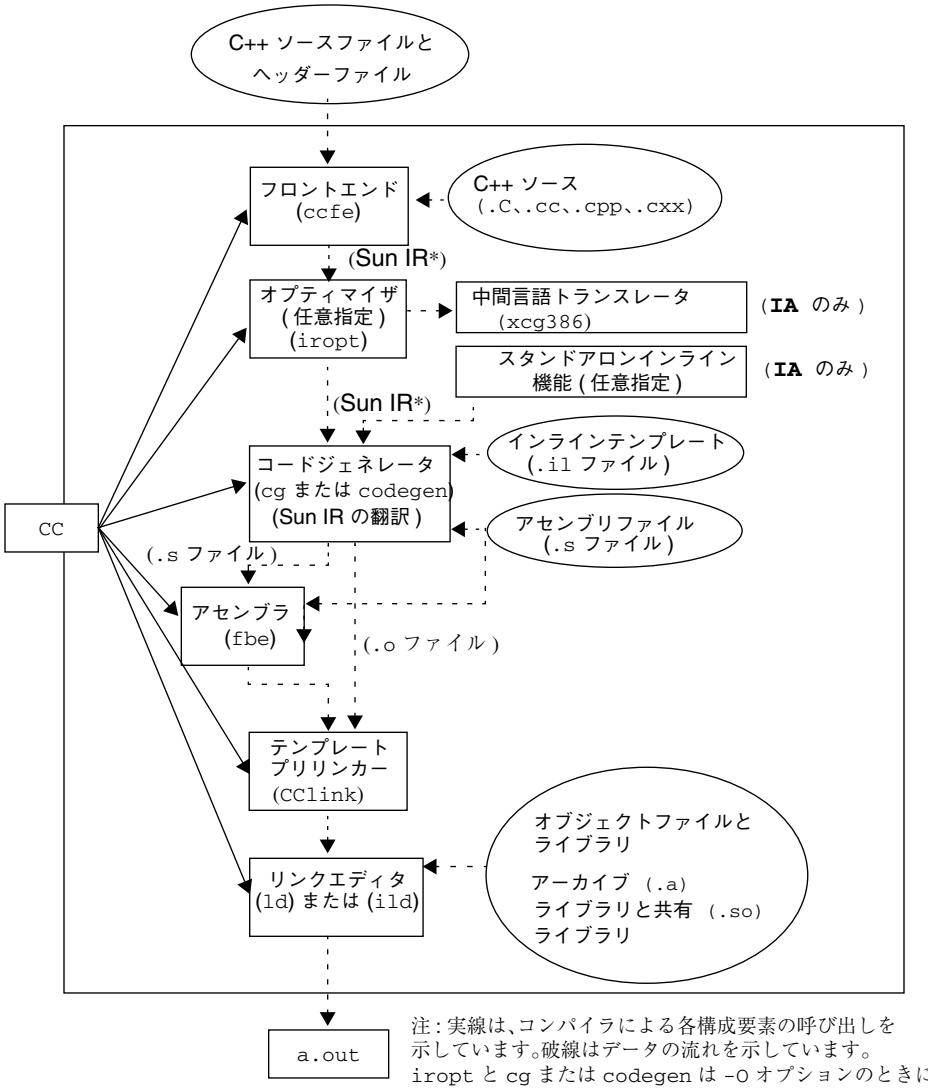

図 2-1　コンパイルの流れ

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	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 (最適化)	-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) 内部処理アナライザ	-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) コード生成	-xO[2-5]、-fast
CClink	テンプレートのリンクの前処理	
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、Sun WorkShop 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。Sun WorkShop 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。

`#define` の変数引数リスト

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
```

マクロ定義の *identifier_list* が省略符号で終わっている場合は、省略符号のほかに、呼び出しの中にマクロ定義内のパラメータよりも多い引数があることを意味します。省略符号表記 inits 引数を使用する、`#define` 前処理指示の交換リストにある識別子 `__VA_ARGS__` を使用してください。詳細は、『C ユーザーズガイド』を参照してください。

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が、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 -x*Olevel* 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。その場合には、最適化レベルを下げるか、複数ルーチンからなるソースファイルを 1 つのルーチンからなるファイルに分割します。

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

標準のライブラリファイルは istream、fstream のような名前で、.h 接尾辞は付いていません。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> がインクルードされている場合、make ユーティリティは istream が実行可能ファイルであるとみなし、istream.cc から istream を構築するときにデフォルトの規則を使用します。このため、非常に誤解を生みやすいエラーメッセージが返されます (istream と istream.cc は両方とも C++ インクルードファイルのディレクトリにインストールされます)。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　オプションの構文形式の例

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

* `option` はオプション名、*value* は値、*pathname* はパス名、*filename* はファイル名を示します。

大括弧、括弧、中括弧、パイプ記号、省略記号は、オプションの説明に使用しているメタキャラクタで、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累積され、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イルする。
`-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`	マルチスレッドコード用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します。
`+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します。
`-xregs`	一時記憶 (スクラッチレジスタ) 用により多くのレジスタを使用できる場合に、コンパイラでのコード生成速度が向上します。このオプションは、追加のスクラッチレジスタを利用可能にします (場合によっては不適切なことがあります)。
`-z` *arg*	リンカーオプション

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 3-3　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	インクルードされたファイルのパス名を出力する。
-keeptmp	コンパイルで作成される一時ファイルを保存する。
-migration	以前のリリ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う。.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*	コンパイラの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。
-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。
-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 3-4　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)* 浮動小数点式の精度を強制しない。
`-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 3-5　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*	さまざまな C++ 言語機能を有効または無効にする。
`-xtrigraphs`	文字表記シーケンスを認識する。

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 3-6　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 lib*lib*.a または lib*lib*.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	マルチスレッドコード用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*	共有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	該当する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。

表 3-6　ライブラリオプション (続き)

オプション	処理
-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。
-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 3-7　ライセンスオプション

オプション	処理
-noqueue	ライセンスの待ち行列化を無効にする。
-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 3-8 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廃止オプション。
`-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 3-9 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。
`-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します。
`-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`	インクルードされたファイルのパス名を出力する。
`-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` ドライバが、型が *sourcetype* の出力を作成する。
`-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 3-9　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。
-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	依存情報を生成するが、/usr/include は除く。
-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。
-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

パフォーマンスオプション

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 3-10　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。
-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します。
-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。

表 3-10 パフォーマンスオプション (続き)

オプション	処理
-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[={0|1)}]	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	*(SPARC)* 最適化された数学ルーチンライブラリを使用する。
-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。
-xO*level*	最適化レベルを *level* にする。
-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。
-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。
-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)* コードサイズを増やす最適化は行わない。
-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。
-xunroll=*n*	可能であればループの最適化を行う。

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 3-11 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 3-12 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	プロファイル処理のため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=tcov	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 3-13 リファレンスオプション

オプション	処理
-migration	以前のコンパイラ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 3-14 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 ファイルの検索パスに *pathname* を追加する。
-I-	インクルードファイル検索規則を変更する。
-xM	makefile 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 3-15 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=*w*	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 3-16 スレッドオプション

オプション	処理
`-mt`	マルチスレッドコード用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。

PART II　C++ プログラムの作成

第4章

言語拡張

-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。

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は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: 名前のない構造体
  struct {
    char* c;
    unsigned d;
  };  // 不正: 名前のない構造体
};
```

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。しかし、Kernighan and Ritchie C (K&R C またはクラシック C)、ANSI C、『Annotated Reference Manual』 の C++ (ARM C++)、および ISO C++ では、1 つのヘッダーファイルの中で、同じプログラム要素に対して複数の宣言や定義を使い分けなければならない場合があります (言語やバージョンによる相違の詳細については、『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++ や ANSI C の宣言
#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"
...                        // ... その他のインクルードファイル ...
#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。べき等を実現する最もよい方法は、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
/* ヘッダーファイルの内容 */
#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。たとえば次のようにします。

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。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); 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 (53 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	`template <class Number> Number twice( Number original );`
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 (53 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。

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、83 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。

```
// 関数パラメータリスト中にテンプレートの型名でない
// パラメータを含む式は、サポートされません。
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。次に例を示します。

```
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 *inital )
    { 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 (62 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。

全クラスインスタンス化

コンパイラは、あるテンプレートクラスを暗黙的にインスタンス化するとき、使用されるメンバーだけをインスタンス化します。コンパイラがあるクラスを暗黙的にインスタンス化するときにすべてのメンバー関数をインスタンス化するには、コンパイラオプションの `-template=wholeclass` を使用します。このオプションを無効にするには、
`-template=no%wholeclass` を指定します。

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>::overflow;
```

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> );
```

そして、次のように定義できます。

```
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、テンプレート引数を省略することができます。次にその例を示します。

```
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 および 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。次に例を示します。

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;`

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

`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<<(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
template class example<simple>
```

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、構築ごとにレ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、83 ページの「テンプレートレポジトリの共有」を参照してください。

makefile a.c、b.c、x.h、x.c を使用した例で説明します。

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

all: a b

a:
    $(CCC) -I.  -c  a.c
    $(CCC) -o a a.o

b:
    $(CCC) -I.  -c  b.c
    $(CCC) -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.c
...
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.c
...
#include "x.h"

main()
{
  X<int> temp1;

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

```
...
b.c
...
#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、テンプレートレポジトリを使って、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。

テンプレートコマンド

テンプレートレポジトリの管理は 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。

テンプレートインスタンスの配置とリンケージ

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すべての開発に適しており、テンプレートのコンパイルとしては総合的に最も優れています。特別な理由がない限り、デフォルトの外部インスタンス方式を使用してください。
- 静的インスタンスは非常に小さなプログラムやデバッグに適しており、用途は限られています。
- 大域インスタンスは、ある種のライブラリ構造に適しています。
- 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、62 ページの「テンプレートのインスタンス化」にあります。

外部インスタンスリンケージ

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レ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。

テンプレートインスタンスは、レ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、外部リンケージで現在のコンパイル単位から参照されます。

外部リンケージは、`-instances=extern` オプションで指定します。このオプションはデフォルトです。

インスタンスはテンプレートレ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。次に例を示します。

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

詳細は、第 6 章を参照してください。

静的インスタンス

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レポジトリに保存されません。

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

大域インスタンスの場合では、すべてのインスタンスが現在のコンパイル単位の中に置かれます。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。テンプレートはテンプレートデータベースに保存されません。

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以外でも認識したり、使用したりできます。その結果、複数のコンパイル単位におけるインスタンス化でリンク作業中に複数のシンボル定義のエラーが生じることがあります。したがって、大域インスタンスは、インスタンスが繰り返されないことがわかっている場合に限り適しています。

大域インスタンスは、`-instances=global` オプションで指定します。

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内に置かれるため、テンプレートは再コンパイルのたびに再インスタンス化され、テンプレートレポジトリには保存されません。

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。同じプログラムで複数の明示的なインスタンス化があると、リンカーで複数シンボル定義エラーになります。したがって、明示的インスタンス方式は、明示的なインスタンス化でライブラリを構成する場合のように、インスタンスが繰り返されないことがわかっている場合に限り適しています。

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。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、テンプレートレポジトリには保存されません。

明示的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。同じプログラムで複数の明示的インスタンス化があると、リンカーで複数のシンボル定義エラーになります。したがって、半明示的インスタンスは、明示的なインスタンス化によってライブラリを構成する場合のように、明示的インスタンスが繰り返されないことがわかっている場合にだけ適しています。

明示的インスタンスの本体内から使用される暗黙的インスタンスは、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の外では認識できません。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、半明示的インスタンスのリンケージは、テンプレート本体で複数のインスタンス化が起こらないプログラムだけに適しています。

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。

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

テンプレートレポジトリ

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。このキャッシュディレクトリは、出力ファイルが置かれる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ので注意してください。

レポジトリの構造

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

テンプレートレ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レ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 にテンプレートインスタンスを ./sub/SunWS_cashe に含まれるレポジトリに書き込みます。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レポジトリからの読み取り

コンパイラは、読み取るオブジェクトファイルに対応したテンプレートレポジトリから読み取りを行います。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
```

テンプレートレポジトリの共有

レ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 (最も保守的で) 最も簡単な方法は、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。

テンプレート定義の検索

定義分離テンプレート編成 (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。したがって、定義検索の必要がない定義取り込み型テンプレートファイル編成を使用するようにしてください。詳細については5 章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。この方法では、テンプレート定義ファイルがテンプレート宣言ファイルと同じベース名を持ち、しかも現在の `include` パスにも存在している必要があります。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、オプションの -pti*directory* (ディレクトリ) 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*directory*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。

テンプレートインスタンスの自動一貫性

テンプレートレポジトリマネージャは、レ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、テンプレートレ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。

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る可能性があります。

オプションファイルの名前は `CC_tmpl_opt` で、`SunWS_config` ディレクトリ内にあります。このディレクトリ名は `SUNWS_CONFIG_NAME` 環境変数で変更できます。`SUNWS_CONFIG_NAME` 変数の値は、パス名ではなく、ディレクトリ名にしてください。

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 行中に納めるようにしてください。分割してはなりません。

コメント

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。

```
# コメント中のテキストは行末まで無視される。
```

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。*options-file* はオプションファイル名を示します。

```
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化であり、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。コンパイル時インスタンス化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。次に例を示します。

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 intd 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。119 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、『The C++ Programming Language』(Third Edition、Bjarne Stroustrup 著、Addison-Wesley、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 { ... };
    // 予期しない例外の処理
    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。

const 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内、const 修飾子を削除できるのは const_cast<> のみ)。T はポインタ、参照、またはメンバー型へのポインタでなければなりません。

```
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_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)。元の型に戻しても、値は元の値と同じになります。
- (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)。次にこの例を示します。

```
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 <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 { };
```

上記の定義の後には次の関数が続きます。

```
void complex_dynamic_casts( )
    {
      AB_B__AB ab_b__ab;
      A*ap = &ab_b__ab;
                    // OK: A を静的に特定できる
      AB*abp = dynamic_cast<AB*>(ap);
                    // 失敗: あいまい
      assert( abp == NULL );
                    // 静的エラー: AB_B* ab_bp = (AB_B*)ap;
                    // 動的キャストではない
      AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // 動的キャストは成功
      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 を static_cast 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、Tom Cargill 著、Addison-Wesley、1992 年発行、『C++ Programming Style』、Jon Louis Bentley 著、Prentice-Hall、1982 年発行、『Writing Efficient Programs』、Dov Bulka と David Mayhew 共著、Addison-Wesley、2000 年発行、『Efficient C++: Performance Programming Techniques』、Scott Meyers 著、Addison-Wesley、1998 年発行、『Effective C++ - 50 Ways to Improve Your Programs and Designs, Second Edition』など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小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	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最も効果的です。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。次に例を示します。

```
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。こうすることで、互換モード (`-compat[=4]`) では `libthread` が必ず `libC` より前にリンクされ、標準モード (デフォルトモード) では `libthread` が必ず `libCrun` より前にリンクされるようになります。

`Dlthread` 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 (93 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。_RWSTDMutex を直接使用。
//

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。_RWSTDGuard を使用。
//

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、131 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しても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(); // ストリーム istr をロック
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // istr のロックを解除
    ...
}
```

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)
```

コード例 11–3 ユーザー定義の入出力操作 (続き)

```
#include <rlocks.h>
#include <iostream.h>
{
    stream_locker sl(this, stream_locker::lock_now);

    *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オーバーヘッドの改善

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);
//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(O4O);
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 オブジェクトをマルチスレッドで使用すると安全ではないに切り換えるには、次のようにします。

コード例 11–5 「マルチスレッドで使用すると安全ではない」への切り換え

```
fs.set_safe_flag(stream_MT::unsafe_object);// マルチスレッドでの安
全性を無効化
    .... 各種の入出力操作を実行
```

iostream が複数のスレッド間で共有されないコード領域では、マルチスレッドで使用すると安全ではないストリームを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) ;
... 各種の入出力操作を実行
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、135 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 { ... };
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();
```

コード例 11–9 新しい関数 (続き)

```
   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);
};

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)
```

コード例 11–9 新しい関数 (続き)

```
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。例として、次のコードを考えてみましょう。

```
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、135 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連の操作を不可分的に実行するためには、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、135 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最も簡単な方法は、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;
```

コード例 11–12 共有オブジェクトの破棄 (続き)

```
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); // スレッドを生成する前に
                                         // fstream オブジェクトを生成
    // スレッドを生成
    for (int i=0; i<numthreads; i++)
        thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);

    ...
    // 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;                          // すべてのスレッドが終了してから
    fp = NULL;                            // fstream オブジェクトを破棄
}
```

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
```

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)

```
// <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;
};

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, O\nO);
    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) {
  // argv: 1 + 各スレッドのファイル名リスト
   if(argc < 2) {
     cout << Ousage: " << argv[0] << " <files..>\n";
```

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)

```
      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)
    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	使用できるモード
`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`	複素数ライブラリ	`-compat=4`
`librwtool`	`Tools.h++ 7.0`	`-compat=4`、`-compat=5`
`librwtool_dbg`	デバック可能な `Tools.h++ 7.0`	`-compat=4`、`-compat=5`
`libgc`	ガベージコレクション	C インタフェース
`libgc_dbg`	デバッグ可能なガベージコレクション	`-compat=4`、`-compat=5`
`libdemangle`	復号化	C インタフェース

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/ja/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。
- 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。
- 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 』(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/ja/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、それ以外は通常通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/ja/man1
- /opt/SUNWspro/man/ja/man3
- /opt/SUNWspro/man/ja/man3C++
- /opt/SUNWspro/man/ja/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「Forte Developer マニュアルページへのアクセス」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/ja/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 -lcx -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 -lcx -lc
```

詳細は、279 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
  - librwtool、librwtool_dbg
  - libgc、libgc_dbg

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
```

本来ならデフォルトで使用される libCrun ライブラリと libCstd ライブラリが、リンクされなくなります。

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) が指定された場合は、-l オプションで明示的に指定されたライブラリだけをリンクします (-xnolib または -nolib が使用された場合は、-library オプションを指定しても無視されます)。

-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。

```
// ファイル 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。

```
// ファイル 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。

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。

```
// ファイル ex1.cc （互換モード）
#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。

```
// ファイル 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)
```

複合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。詳細は、279 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++ バージョン 7	-compat=4	-library=rwtools7
	-compat=5	-library=rwtools7,iostream
		-library=rwtools7_std

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 (続き)

ライブラリ	コンパイルモード	オプション
デバッグ対応 Tools.h++ バージョン 7	-compat=4 -compat=5	-library=rwtools7_dbg -library=rwtools7_dbg,iostream -library=rwtools7_std_dbg
ガベージコレクション	-compat=4 -compat=5	-library=gc -library=gc
デバッグ対応ガベージコレクション	-compat=4 -compat=5	-library=gc_dbg -library=gc_dbg

標準ライブラリの静的リンク

デフォルトでは、CC ドライバは、デフォルトライブラリのそれぞれについて -l*lib* オプションをリンカーに渡すことによって、libc や libm などの共有ライブラリをいくつか静的にリンクします (互換性モードと標準モードのデフォルトライブラリについては、149 ページの「デフォルトの C++ ライブラリ」を参照)。

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。次に例を示します。

```
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。詳細は、335 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.1
- libC.so.5
- libcomplex.so.5
- librwtool.so.2
- libgc.so.1
- libgc_dbg.so.1
- libCstd.so.1
- libiostream.so.1
- libCstd.so
- libiostream.so

注 – libCstd と libiostream の共有ライブラリを使用するときは、必ず 186 ページの「共有版の libiostream」の手順に従ってください。

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.1: a.out: libcomplex.so.5: open に失敗しました :
ファイルもディレクトリもありません。
```

ただし、環境変数 LD_BINARY_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/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/lib:${LD_LIBRARY_PATH}
example$ export LD_LIBRARY_PATH
```

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++ の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

コンパイラに添付されている標準ライブラリの代わりに別の標準ライブラリを使用することは危険で、必ずしもよい結果には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。

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。次に例を示します。

```
example% CC -I/opt/mycstd/include -library=no%Cstd ...(コンパイルの場合)
```

-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。次に例を示します。

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd ... (リンクの場合)
```

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。次に例を示します。

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a ... (リンクの場合)
```

-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。Solaris 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 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 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

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 < > に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 の string.h
<fcntl.h>	fcntl.h	標準 C および C++ヘッダー以外
"string"	string	山かっこではなく、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

160 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、159 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 Standard C++ Library、Version 2 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 4.5.2 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、183 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。詳細については、158 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++ Class Library Reference』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」 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 「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 のマニピュレータ
<ios>	iostream の基底クラス
<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
_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

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に対する読み書きをサポートする
basic_ios	すべてのストリームが共通に必要とする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

マニュアルページ	概要
basic_iostream	ストリームバッファ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が制御する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に対する書き込みをサポートする
basic_ostream	ストリームバッファ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<chartT,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

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 stderr に 関連付けられたストリームバッファからの入力を 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

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 stdout に関連付けられたストリームバッファ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

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
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
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

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
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
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

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
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
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

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
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
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

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
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
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<chartT,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

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
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
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

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
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
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

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
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
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_trais<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

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
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
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

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
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
wfilebuf	入力または出力シーケンスをファイルに関連付ける
wfstream	ファイル記述子に関連付けられた指定ファイルまたはその他デバイスに対する読み取りと書き込みをサポートする
wifstream	ファイル記述子に関連付けられた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共通に必要とする関数を取り込む基底クラス
wistream	ストリームバッファーが制御する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	メモリー上の配列への basic_string<chartT,traits,Allocator> クラスのオブジェクト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

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
wostringstream	basic_string<chartT,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_trais <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、279 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 4.5.2 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 4.5.2 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。
- 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。
- 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)。

  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、入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
```

共有版の libiostream

C++ コンパイラには、従来の iostream ライブラリ、libiostream の共有版が付属しています。

共有版の libiostream を使用するには以下を実行します。

1. スーパーユーザーとして、次のシンボリックリンクを手作業で作成します。

```
example% ln -s /usr/lib/libiostream.so.1 \
  /opt/SUNWspro/lib/libiostream.so
example% ln -s /usr/lib/sparcv9/libiostream.so.1 \
  /opt/SUNWSpro/lib/v9/libiostream.so
```

IA プラットフォームでは、最後のリンクは必要ではありません。

注 – コンパイラが /opt/SUNWSpro ディレクトリにインストールされていない場合は、システム管理者にインストール先のディレクトリを確認してください。

---

2. シンボリックリンクが正しく作成されたかどうかを確認します。

-library=iostream オプションを付けて /opt/SUNWSpro/bin/CC を起動し、任意のプログラムをコンパイルします。コンパイルが完了した後で ldd a.out と入力し、出力を参照して /usr/lib/libiostream.so.1 がリンクされているかどうかをチェックします。

3. これらのシンボリックリンクを作成すると、-library=iostream オプションを使用した場合に、デフォルトで libiostream が動的にリンクされます。

このライブラリを静的にリンクするには、 -library=iostream -staticlib=iostream オプションを使用します。

注 – 共有版の libiostream ライブラリにリンクしたオブジェクトファイルを第三者に配布する場合は、配布する製品に最新の SUNWlibC パッチを添付するか、配布先の顧客に http://sunsolve.sun.com などの Sun の Web サイトから、最新の SUNWlibC パッチを入手してもらう必要があります。このパッチは無料で、自由に再配布することができます。

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。189 ページの「iostream を使用した出力」と 193 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。この他に、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、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

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。
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	(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。この中で iostream.h、fstream.h、iomanip.h、stdiostream.h をインクルードします。

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分かれば、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、197 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。

```
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。

```
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 (203 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。次の例のように、挿入演算子と同様に繰り返し指定することができます。

```
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;
    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。

```
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。使用例を次に示します。

```
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 <unistd.h>
#include <iostream.h>
void error (const char* message) {
    cout << message << "\n" ;
    exit(1);
}
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。208 ページの「streambuf」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分かっている場合は、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。次に例を示します。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::end);
```

一方、次の例では現在位置から 10 バイト進められます。

```
aFile.seekp(10, ios::cur);
```

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 204 ページの表 14-2 「iostream の定義済みマニピュレータ」 で説明します。表 14-2 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。
- m 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。

1. ストリームの参照引数を受け取ります。
2. そのストリームに何らかの処理を行います。
3. 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);
   }
```

strstream： 配列用の iostream

strstream(3CC4) のマニュアルページを参照してください。

stdiobuf: 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。

streambuf

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(3CC4) 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 >> (streambuf*)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。 - istream クラスに対するメンバー関数で、streambuf から取り 出した文字の解釈をサポートする - 入力のフォーマット - ostream クラスで記述されている位置決め関数 - その他の istream に関連した関数 - istream に関連したマニピュレータ
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。 - ostream クラスに対するメンバー関数で、streambuf に書き込まれた文字の解釈をサポートする - 出力のフォーマット - ostream クラスで記述されている位置決め関数 - その他の ostream に関連した関数 - ostream に関連したマニピュレータ
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

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。

表 14-4 iostream の用語 (続き)

用語	意味
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。
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。

```
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;
```

引数は 1 つまたは 2 つ指定することができ、どちらの場合も 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);
```

上の例では、次の値を持つ複素数の変数が生成されます。

```
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);
```

上の例では、次の値を持つ複素数の変数が生成されます。

```
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 (218 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。

この他、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次の 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。

`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)`
	この関数は引数を 2 つ持ちます。 pow(*b*, *exp*) とすると、 *b* の *exp* 乗が計算されます。

`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。

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 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

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、187 ページの「iostream 操作の基本構造」と 189 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、

(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。

```
complex x;
cin >> x;
```

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。次の例で考えてみます。

```
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) は、引数の型が一致していない
```

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 型のものの他に、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、次のように最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、最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。構築の例を次に示します。

```
% 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、テンプレートレ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レ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は、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。
- 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` 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 または 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
- 実行時型特定機構 (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、-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、および 8 で実行する cc コンパイラのコマンド行オプションを詳しく説明します。ここで説明する機能は、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 として特定され、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	内容
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。
展開	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。たとえば「`-xO` が 3 より小さい場合は、`-xinline` オプションを使用すべきではありません」などです。

表 A-2　オプションの見出し (続き)

見出し	内容
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	シンボルが他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 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 -B*binding* 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)、155 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*

-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 199711L 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 0x540 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
- -library=[no%]strictdestrorder
- -library=[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 -Bsymbolic 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

タイプ	マクロ名	注
SPARC and IA	__ARRAYNEW	「配列」形式の operator new と operator 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=0x540	__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	240 ページの「-compat [={4|5}]」を参照。
	__SVR4	
	__TIME__	

表 A-3　SPARC と IA 用の事前定義シンボル (続き)

タイプ	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	

デフォルト

=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 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

(SPARC) 可能な場合には、ダブルワードのロードとストア命令を生成してパフォーマンス向上を図ります。

このオプションは、double 型のデータがすべて double の境界から始まることを前提としています。

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、次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);
}
```

このプログラム結果は次のようになります。

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』

-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-4　-fast 展開

オプション	SPARC	IA
-dalign	○	−
-fns	○	○
-fsimple=2	○	−
-ftrap=%none	○	○
-nofstore	−	○
-xarch	○	○
-xlibmil	○	○
-xlibmopt	○	○

表 A-4　-fast 展開

オプション	SPARC	IA
-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。

```
examle% 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

-dalign、-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、*a* に次の値の 1 つを指定できます。

表 A-5　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。無効にした場合、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-5　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。94 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-5　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。-feature=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-6　標準モードだけに使用できる -features オプション

[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-7　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。
[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。

表 A-7　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%]rtti	実行時の型識別 (RTTI) を許可します [しません]。dynamic c_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
```

- 標準モード (デフォルトモード)

```
-features=%all,no%iddollar
```

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

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。

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*...]]

コンパイラにより、リンカーエラーメッセージに通常適用されるフィルタリングを抑止します。

filter は次の値のいずれである必要があります。

表 A-8　filt オプション

filter の値	意味
[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%]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。
%all	-filt=errors,names,returns 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 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=errors、names,returns 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=names,returns,errors が使用され、標準出力が表示されます。

```
example% CC filt_demo.cc
未定義の                              最初に参照している
シンボル                                  ファイル
type::~type()          filt_demo.o
type::__vtbl           filt_demo.o

ld: 重大なエラー: シンボル参照エラー。a.out に書き込まれる出力はありません。
```

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
未定義の                              最初に参照している
シンボル                                  ファイル
__1cEtype2T6M_v_                       filt_demo.o
__1cEtypeG__vtbl_                      filt_demo.o
ld: 重大なエラー: シンボル参照エラー。a.out に書き込まれる出力はありません。
```

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。

-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[={no|yes}]

(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。

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 には次のいずれかを指定します。

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 には次のいずれかを指定します。

r の値	意味
`nearest`	最も近い数値に丸め、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。

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。

このオプションを無効にするには、`-nofstore` オプションを使用します。

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。

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

このオプションを使用してルーチンを 1 つコンパイルした場合は、プログラムのルーチンもすべて同じオプションを使用してコンパイルしてください。そうしないと、予期しない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 `-1` 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 `-1` 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、`-1Crun` 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、第16章の227 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、339 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』の「データ収集と解析のためのアプリケーションのコンパイル」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

インクルードされるファイルのパス名を出力します。

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。

コンパイル時のローダーは、指定された名前を作成中の動的共有ライブラリに割り当て、そのライブラリのイントリンシック名 (固有名) としてライブラリの中に記録します。-h*name* (名前) オプションを指定しないと、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、および特殊な目的の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、160 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。158 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++ ANSI C ヘッダファイル、および特殊な目的のファイル内ヘッダファイルの include ヘッダーファイル
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、160 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。

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、それらに対して静的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=extern が使用されます。

関連項目

第 7 章、77 ページの「テンプレートのコンパイル」

-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、デバックに便利です。

関連項目

`-v`、`-verbose`

-KPIC

(SPARC) `-xcode=pic32` と同じです。

(IA) `-Kpic` と同じです。

このオプションは、共有ライブラリの構築時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介して 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。大域データへの各参照は、大域オフセットテーブルにおける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。

-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-9 互換モードでの -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	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%]gc_dbg	デバッグ可能なガベージコレクション libgc を使用します [しません]。

表 A-9　互換モードでの -library オプション (続き)

l 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、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」の節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-10 標準モードでの -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 iostream 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]。

表 A-10 標準モードでの -library オプション (続き)

l の値	意味
[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%]gc_dbg	デバッグ可能な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 4.5.2 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、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」の節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 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` および `-library=Cstd` を同一のコマンド行で使用することはできません。

同時に使用できる Rogue Wave ツールライブラリは 1 つだけです。また、`-library=stlport4` を指定して Rogue Wave 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 Rogue Wave 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 Rogue Wave 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、Rogue Wave `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         <-- 有効
% CC -compat -library=rwtools_std foo.cc     <-- 無効

% CC -library=rwtools,iostream foo.cc        <-- 有効、従来の iostream
% CC -library=rwtools foo.cc                 <-- 無効

% CC -library=rwtools_std foo.cc             <-- 有効、標準 iostream
% CC -library=rwtools_std,iostream foo.cc    <-- 無効
```

`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。

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分からない。
- 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、26 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++ Class Library 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、158 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
- 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*

`-xO`*level* 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レ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` 内のレ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レ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、82 ページの「テンプレートレポジトリ」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

レポジトリのディレクトリについては、82 ページの「テンプレートレポジトリ」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。

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。たとえば、ccfe に -features=bool、iddollar を渡そうと次のように指示するとします。

```
    -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。

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_dirctory*/SUNWspro/lib/v9
上記以外の値	*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。

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 オプションで指定されているライブラリと、-xlang オプションで指定されているライブラリ、-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`、155 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。

opt の値	意味
[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。
[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]。

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。

関連項目

第 6 章の「全クラスインスタンス化」、第 7 章の 83 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 つめ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*...]

コンパイラの冗長性を制御します。

値

v には、次に示す値の 1 つを指定します。

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。
[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

−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。つまり、-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からなるプログラムをリンクすることはできますが、両方のオプションを使って 1 つのファイルを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

  基本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。以下の基本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	列挙型
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。

-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。

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

第 16 章「ライブラリの構築」

-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-11 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-11 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、この値は -xarch=v7 に相当します。
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最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最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。 例: 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-11 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-11 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=v9a は、64 ビット対応 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。
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-III 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-12 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-12 IA プラットフォームでの -xarch 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	このリリースでは、486 と pentium_pro は同じです。

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=v8 plusa 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の指定と一緒に使用する場合、適切な選択をすれば、指定したアーキテクチャで実行可能ファイルの良好なパフォーマンスが得られます。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。

-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) オプティマイザで使用するキャッシュ属性を定義します。

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。

c の値	意味
generic	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るキャッシュ属性を定義します。
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* を意味します。

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 の設定内容は、次のとおりです。

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
ラインサイズ 32 バイト	ラインサイズ 32 バイト
4 ウェイアソシアティブ	ダイレクトマッピング

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。

-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-13 -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。

- 命令の順番 (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-14 `-xchip` オプション

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 プロセッサより以前のプロセッサのタイミング属性プロセッサ
	`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

表 A-14 -xchip オプション

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
	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。

値

a には次の値のいずれかを指定します。

表 A-15 -xcode オプション

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 + bss サイズは 2**32 バイトに限定されます。
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 + bss サイズは 2**44 バイトに限定され、64 ビットアーキテクチャ -xarch={v9|v9a|v9b} でのみ使用可能です。
abs64	(SPARC) 64 ビット絶対アドレスを生成します。低速ですが全範囲を使用でき、64 ビットアーキテクチャ -xarch={v9|v9a|v9b} でのみ使用可能です。
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 と同等。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 と同等。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。

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) 複数のソースファイルに渡る最適化とインライン化を可能にします。-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。

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。

-xF

この -xF オプションを指定してコンパイルした後で実行してアナライザを使用すると、最適化された関数の順序を示すマップファイルを作成できます。続いて実行するリンカーには、-M*mapfile* (マップファイル) オプションでそのマップを使用するよう指示して、実行可能ファイルを作成することができます。これによって、実行可能ファイルの各関数が別々のセクションに置かれます。

メモリー上でサブプログラムの順序を並べ替えることで効果が上がるのは、アプリケーション時間の多くの割合がアプリケーションテキストのページフォルト時間に費やされている場合だけです。それ以外の場合は、順序を変えてもアプリケーションの全体的なパフォーマンスが向上しないことがあります。

相互の関連性

-xF オプションは、-features=no%except (-noex) のときにだけ有効です。

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 と、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-16 -xinline オプション

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 -x03 未満の最適化レベルには影響を与えません。-x04 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。

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 -x03 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

-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。

-xipo[={0|1}]

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。

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。

-xlang=*language*[, *language*]

指定された言語に対して該当する実行時ライブラリを取り込み、正しい実行時環境を整えます。

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

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 Libraries 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`

-xlibmieee

例外時に `libm` が数学ルーチンに対し IEEE 754 値を返します。

`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最も高速な実行可能コードが生成されます。

相互の関連性

この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 ... ← 推奨
```

-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。-Xm

-features=iddollar と同じです。

-xM

makefile の依存情報を出力します。

例

プログラム foo.cc には次の文が含まれています。

```
#include "foo.h"
```

foo.c を -xM でコンパイルすると、次の行が出力に含まれます。

```
foo.o:foo.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 ヘッダファイルの依存関係とコンパイラで提供されるヘッダ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) データセグメントをテキストセグメントと併合 (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

-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、C++ 標準ライブラリを Java コードから呼び出すことができるように、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 の使用方法の詳細については、Forte for Java Enterprise Edition の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。

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。

相互の関連性

-g と -xnativeconnect の両方を指定するとプログラムがコンパイルされない場合は、-g オプションを指定せずにコンパイルしてください。

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、-lcs 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
```

-xopenmp[=*i*]

(SPARC) C++ コンパイラは、明示的な並列化用の OpenMP インタフェースを実装しています。ソースコード指令、実行時ライブラリルーチン、環境変数を次のオプションで指定します。

- -xopenmp[=*i*]

ここで、*i* には parallel、stubs、none のいずれかを指定します。

-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 だけを指定した場合は、コンパイラでは -xopenmp=parallel が設定されます。このオプションは OpenMP プラグマを認識し、SPARC だけに適用されます。-xopenmp=parallel での最適化レベルは -x03 です。コンパイラは、プログラムの最適化レベルが -x03 未満から -x03 に変更された場合に警告を出力します。

-xopenmp=stubs コマンドは、OpenMP API ルーチンの stubs ルーチンにリンクします。このオプションは、アプリケーションを逐次実行するようにコンパイルする必要がある場合に使用します。-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。

-xopenmp=none コマンドは、OpenMP のプラグマの認識を有効にせず、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。

関連項目

『*OpenMP API User's Guide*』

-xO*level*

最適化レベルを指定します。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、実行速度が速くなります。

-xO*level* を指定しないと、非常に基本的なレベルの最適化しか行われません。つまり、最適化は、式の局所的な共通部分を削除することと、デッドコードを分析することに限定されます。最適化レベルを指定してコンパイルすると、プログラムのパフォーマンスが著しく向上することがあります。ほとんどのプログラムの場合、-xO2 (または同等のオプション -O および -O2) を使用することをお勧めします。

一般に、プログラムをより高い最適化レベルでコンパイルすれば、実行時のパフォーマンスはそれだけ向上します。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 IA 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 -xO3 を使用すると、コードサイズが大きくなります。

- -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、最高レベルで最適化が行われます。これを使用するのは、コンピュータの最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。347 ページの「–xprofile=p」 を参照してください。

IA 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った場合、通常は実行速度が早くなりますが、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。

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。

実際のスワップ空間に関する情報を得るには、**swap -l** と入力します。

実際の実メモリーに関する情報を得るには、**dmesg | grep mem** と入力します。

関連項目

-fast、-xcrossfile=*n*、-xprofile=*p*、csh(1) のマニュアルページ

–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』

–xprefetch[=*a*[,*a*]]

(SPARC) 先読み機能をサポートするアーキテクチャで先読み命令を有効にします。たとえば、UltraSPARC-II (-xarch=v8plus、v8plusa、v8plusb、v9、v9a、v9b のいずれか) の場合です。

a は次のどれかです。

表 A-17 -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。

表 A-17 -xprefetch の値 (続き)

値	意味
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なさることをお勧めします。

-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-18 -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*

実行時プロファイルデータを収集したり、それを使って最適化します。

このオプションを使用すると、実行頻度のデータが集められて、実行時に保存されます。保存されたデータは後続する処理の実行時に使用され、これによってパフォーマンスが向上します。このオプションは、最適化のレベルが指定されている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。

表 A-19 -xprofile オプション

p の値	意味
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* は省略可能です。指定しなかった場合、a.out と仮定されます。 -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。 • 注: -xprofile=collect を指定して共有ライブラリをコンパイルすることはできません。
use[:*name*]	有効な最適化を行うために実行頻度データを使います。*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。 プログラムは、前の実行で feedback ファイルに生成され、保存された実行頻度データを使って最適化されます。このファイルは、-xprofile=collect でコンパイルしたプログラムを前に実行したときに書き込まれたものです。 ソースファイルと他のコンパイラオプションは、feedback ファイルを生成したコンパイル済みプログラムをコンパイルしたときとまったく同じでなければなりません。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* でコンパイルしたのであれば、同じプログラム名 *name* を最適化コンパイルの -xprofile=use:*name* にも指定しなければなりません。

表 A-19 -xprofile オプション (続き)

p の値	意味
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 オプションとは異なり、TCOVDIR 環境変数はコンパイル時間には影響しません。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 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。つまり、-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からなるプログラムをリンクすることはできますが、両方のオプションを使って 1 つのファイルを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』

-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。各値の意味は `-xarch` の設定によって異なります。

r の値	内容
`[no%]appl`	`V8` および `V8a` の場合、レジスタ %g2、%g3、%g4 の使用を許可します [しません]。 `v8plus`、`v8plusa`、および `v8plusb` の場合、レジスタ g2、g3、g4、g5 の使用を許可します [しません]。 `V9`、`V9a`、および `v9b` の場合、レジスタ %g2 と %g3 の使用を許可します [しません]。 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`	SPARC ABI で指定されているように、浮動小数点レジスタの使用を許可します [しません]。 プログラム中に浮動小数点コードが含まれていない場合でも、これらのレジスタを使用できます。 浮動小数点コードが含まれているソースプログラムには、このオプションを使用できません。

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、次のように指定します。

```
-xregs=appl,float
```

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、次のように指定します。

```
-xregs=no%appl, no%float
```

関連項目

SPARC V7 および V8 の ABI、SPARC V9 の 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。また、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生成するだけで、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。

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-20 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、コンパイラが動作しているマシンで使用できるアーキテクチャ、チップ、キャッシュの属性を決定します。
`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 `-xarch=generic -xchip=generic -xcache=generic` に展開します。 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。 32 ページの表 3-6 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-21 -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
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

表 A-21 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
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
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

表 A-21 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
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
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

表 A-21 -xtarget の SPARC プラットフォーム名 (続き)

-xtarget	-xarch	-xchip	-xcache
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
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	64/32/4:8192/256/1
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-22　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、および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-23 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。

展開

-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 自体の意味は、展開することです。

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 オプションを指定して SPARC V9 アーキテクチャ用にコンパイルする場合、-xtarget=ultra または ultra2 の指定は必要でないか、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。たとえば、次のように指定するとします。

```
-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。

```
–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。

-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。

-xunroll=*n*

可能な場合は、ループを展開します。

このオプションでは、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。

-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

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いと、コンパイラでは -xtrigraphs=yes が使用されます。

-xtrigraphs だけを使用すると、コンパイラでは -xtrigraphs=yes が使用されます。

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。

-xvector[=(yes|no)]

(SPARC) SPARC ベクトルライブラリ関数の自動呼び出しを有効にします。

デフォルト

コンパイラのデフォルトは -xvector=no です。-xvector だけを指定した場合、-xvector=yes が仮定されます。

警告

コンパイルとリンクを別々に行う場合は、どちらにも同じ -xvector 設定を使用する必要があります。

-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。

-ztext

書き込み不可で割り当て可能なセクションに対して再配置が残っている場合に、致命的エラーとします。

このオプションはリンカーに渡されます。

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 pragma は C++ 標準の一部です。ただし、プラグマの書式、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。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。

注 – 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。

Sun WorkShop C++ コンパイラが認識する一般的なプラグマのキーワードを次に示します。

- `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。
- `init` - 指定した関数を初期化関数にします。
- `fini` - 指定した関数を終了関数にします。
- `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。
- `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。
- `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。

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。

#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 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 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 #pragma で指定する前に宣言しておく必要があります。

#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 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、現在のコンパイル単位に含まれていなければなりません。また、このプラグマは、指定する関数と同じスコープに含まれていなければならず、その関数の宣言と定義の間に配置する必要があります。

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 weak 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最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最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最も厳密な境界整列

プラットフォーム	最も厳密な境界整列
IA	4
SPARC 一般、V7、V8、V8a、V8plus、V8plusa	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
float	4、4	4、4	4、4
double	8、8	8、8	8、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
long double	16、8	16、16	12、4
データへのポインタ	4、4	8、8	4、4
関数へのポインタ	4、4	8、8	4、4
メンバーデータへのポインタ	4、4	8、8	4、4
メンバー関数へのポインタ	8、4	16、8	8、4

#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。この情報は、オプティマイザがポインタ値を効率的に管理したり、メモリー上の位置を明確にするのに役立てられ、その結果としてスケジューリングやパイプライン処理が効率的になります。

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。また、このプラグマは、指定する関数と同じスコープに含まれていなければならず、その関数の宣言と定義の間に配置する必要があります。

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 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。

#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。

他のコンパイル単位に関数または変数の強い (strong) 定義が存在する場合、*name* はその定義にリンクされます。*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。次に例を示します。

```
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_" // float bar(int) を弱いシンボルにする
```

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。

事前束縛

「動的束縛」を参照。

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。

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つけること。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つけて、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

<< 挿入演算子
  complex, 221
  iostream::, 189, 191
! 演算子、ios::, 192
$識別子最初以外の文字として許可, 253
>> 抽出子演算子
  complex, 221

数字

-386、コンパイラオプション, 237
-486、コンパイラオプション, 237

A

ABI (アプリケーションバイナリインタフェース)、ライブラリの構築, 229
abs、complex::, 219
accumulate のマニュアルページ, 169
acos、complex::, 219
adjacent_difference のマニュアルページ, 169
adjacent_find のマニュアルページ, 169
advance のマニュアルページ, 169
Algorithms のマニュアルページ, 168
allocator のマニュアルページ, 169
ang、complex::, 219
__ARRAYNEW、事前定義マクロ, 244
asin、complex::, 219
Associative_Containers のマニュアルページ, 168
atan、complex::, 219
auto_ptr のマニュアルページ, 169
.a、ファイル名接尾辞, 225
-a、コンパイラオプション, 237
.a、ファイル名接尾辞, 11

B

back_inserter のマニュアルページ, 169
back_insert_iterator のマニュアルページ, 169
badbit、ios::, 192
basic_filebuf のマニュアルページ, 169
basic_fstream のマニュアルページ, 169
basic_ifstream のマニュアルページ, 169
basic_iostream のマニュアルページ, 170
basic_ios のマニュアルページ, 169
basic_istream のマニュアルページ, 170
basic_istringstream のマニュアルページ, 170
basic_ofstream のマニュアルページ, 170
basic_ostream のマニュアルページ, 170

basic_ostringstream のマニュアルページ, 170
basic_streambuf のマニュアルページ, 170
basic_stringbuf のマニュアルページ, 170
basic_stringstream のマニュアルページ, 170
basic_string のマニュアルページ, 170
-B*binding*, compiler option, 97
-b*binding*、コンパイラオプション, 32, 238 ~ 240
Bidirectional_Iterators のマニュアルページ, 168
binary_function のマニュアルページ, 170
binary_negate のマニュアルページ, 170
binary_search のマニュアルページ, 170
bind1st のマニュアルページ, 170
bind2nd のマニュアルページ, 170
binder1st のマニュアルページ, 171
binder2nd のマニュアルページ, 171
bitset のマニュアルページ, 171
_BOOL、事前定義マクロ, 244
__BUILTIN_VA_ARG_INCR、事前定義マクロ, 244

C

C API (アプリケーションプログラミングインタフェース)、ライブラリの構築, 230
  C++ 実行時ライブラリにおける依存関係を削除, 230
C 標準ライブラリヘッダーファイル、置き換え, 162
C++ 標準ライブラリ, 146 ~ 147
  RogueWave 版, 166
  置き換え, 158 ~ 163
  構成要素, 165
  マニュアルページ, 148, 168
C++ マニュアルページ、アクセス, xxxiii, 148, 149
.c++、ファイル名接尾辞, 10
cartpol、complex のマニュアルページ, 224
CC プラグマ指令, 365
CCadmin コマンド, 78
ccfe、コンパイル構成要素, 19
CCFLAGS、環境変数, 24
CClink、コンパイル構成要素, 19
CC_tmpl_opt オプションファイル, 85
.cc、ファイル名接尾辞, 10
cerr
  iostream::, 187
  マニュアルページ, 171
cerr 標準ストリーム, 134
c_exception、定義, 220
-cg[89|92]、コンパイラオプション, 240
cg、コンパイル構成要素, 19
char*、抽出子, 194 ~ 195
char_traits のマニュアルページ, 171
cin
  iostream::, 187
  マニュアルページ, 171
cin 標準ストリーム, 134
clog
  事前定義 iostreams, 187
  マニュアルページ, 171
clog 標準ストリーム, 134
codecvt_byname のマニュアルページ, 171
codecvt のマニュアルページ, 171
collate_byname のマニュアルページ, 171
collate のマニュアルページ, 171
compare のマニュアルページ, 171
-compat
  C++ ライブラリのリンクのモード, 154
  -features オプション、値の制限, 252
  デフォルトのリンク済みライブラリでの影響, 149
  ライブラリ、使用可能なモード, 146
-compat、コンパイラオプション, 29, 31, 240
complex
  エラー処理, 219 ~ 221
  演算子, 218
  型 complex, 216
  効率, 223

混合モード, 223
コンストラクタ, 216, 217
数学関数, 220, 218
入出力, 221
複素数ライブラリ, 215 ~ 216
マニュアルページ, 224, 171
ライブラリ, 153 ~ 154, 146 ~ 147
complex()、コンストラクタ, 217
complex.h、complex ヘッダーファイル, 216
complex_error
定義, 220
メッセージ, 218
conj、complex::, 219
Containers のマニュアルページ, 168
copy_backward のマニュアルページ, 171
copy のマニュアルページ, 171
cos、complex::, 219
cosh、complex::, 219, 221
count_if のマニュアルページ, 171
count のマニュアルページ, 171
cout
iostream::, 187, 190
マニュアルページ, 172
cout、標準ストリーム, 134
__cplusplus、事前定義マクロ, 51, 244
cplx.intro、complex のマニュアルページ, 224
cplxerr、complex のマニュアルページ, 224
cplxexp、complex のマニュアルページ, 224
cplxops、complex のマニュアルページ, 224
.cpp、ファイル名接尾辞, 10
ctype_byname のマニュアルページ, 172
ctype のマニュアルページ, 172
.cxx、ファイル名接尾辞, 10
-c、コンパイラオプション, 14, 34, 240
.C、ファイル名接尾辞, 10
.c、ファイル名接尾辞, 10

D

-dalign、コンパイラオプション, 246
__DATE__、事前定義マクロ, 244
-DDEBUG, 84
definition *name*、コンパイラオプション, 87
delete 列形式、認識, 255
deque のマニュアルページ, 172
_distance_type のマニュアルページ, 169
distance のマニュアルページ, 172
divides のマニュアルページ, 172
dlclose()、関数呼び出し, 227
dlopen(), function call, 229
dlopen()、関数呼び出し, 226, 231
dmesg、実際のメモリー, 23
-D_REENTRANT, 126
-dryrun、コンパイラオプション, 30, 34, 247
dynamic_cast 演算子, 102
-D、コンパイラオプション, 28, 37, 243 ~ 245
+d、コンパイラオプション, 30, 242
-d、コンパイラオプション, 32, 243, 246

E

-E
出力オプション, 34
デバッグオプション, 30
プリプロセッサオプション, 37
+e(0|1)、コンパイラオプション, 29, 248
EDOM、errno 設定, 221
enum
スコープ修飾子、enum 名の使用, 45
前方宣言, 44
不完全、使用, 45
eofbit、ios::, 192
equal_range のマニュアルページ, 172
equal_to のマニュアルページ, 172
equal のマニュアルページ, 172
ERANGE、errno 設定, 221
errno、定義, 221
error、iostream::, 192

exception のマニュアルページ, 172
exp、complex::, 219 ~ 221
explicitキーワード、認識, 255
export キーワード、 認識, 253
-Eオプションの説明, 247, 248

F

facets のマニュアルページ, 172
failbit、ios::, 192
-fast、コンパイラオプション, 14, 35, 249 ~ 251
fbe、コンパイル構成要素, 19
-features、コンパイラオプション, 43, 94, 31
filebuf
  マニュアルページ, 172
__FILE__、事前定義マクロ, 244
fill_n のマニュアルページ, 172
fill のマニュアルページ, 172
find_end のマニュアルページ, 172
find_first_of のマニュアルページ, 172
find_if のマニュアルページ, 172
find のマニュアルページ, 172
-flags、コンパイラオプション, 260
float 型挿入子、iostream 出力, 189
-fnonstd、コンパイラオプション, 260
-fns[=(yes|no)]、コンパイラオプション, 31
-fns、コンパイラオプション, 260
for_each のマニュアルページ, 173
Fortran 実行時ライブラリ、, 329
Forward_Iterators のマニュアルページ, 168
fpos のマニュアルページ, 173
-fprecision=*p*、コンパイラオプション, 31, 262
front_inserter のマニュアルページ, 173
front_insert_iterator のマニュアルページ, 173
-fround=*r*、コンパイラオプション, 31, 263 ~ 264
-fsimple=*n*、コンパイラオプション, 31, 264
-fstore、コンパイラオプション, 31, 266
fstream
  iostream::, 188
  マニュアルページ, 173
fstream.h
  iostream ヘッダーファイル, 189
-ftrap、コンパイラオプション, 31, 266
Function_Objects のマニュアルページ, 168
__func__、識別子, 49

G

-G
  オプションの説明, 268 ~ 269
  出力オプション, 34
  動的ライブラリコマンド, 227 ~ 228
  ライブラリオプション, 32
-g
  オプションの説明, 269
  コード生成オプション, 29
  コンパイルとリンク, 14
  デバッグオプション, 30
  テンプレートコンパイルオプション, 84
generate_n のマニュアルページ, 173
generate のマニュアルページ, 173
get、char 抽出子, 195
get_temporary_buffer のマニュアルページ, 173
-gO
  オプションの説明, 269
  コンパイラオプション, 14
  デバッグオプション, 30
goodbit、ios::, 192
good、ios::, 192
gprof、C++ ユーティリティ, 6
greater_equal のマニュアルページ, 173
greater のマニュアルページ, 173
gslice_array のマニュアルページ, 173
gslice のマニュアルページ, 173
-g、コンパイラオプション, 35

H

-H
  プリプロセッサオプション, 37, 38
hardfail、ios::, 192
has_facet のマニュアルページ, 173
Heap_Operations のマニュアルページ, 168
-help、コンパイラオプション, 272
-h*name*、コンパイラオプション, 32, 271
-H、コンパイラオプション, 30, 34, 271

I

-I-
  プリプロセッサオプション, 38
__i386、事前定義マクロ, 245
i386、事前定義マクロ, 245
IAの定義, 19
ifstream
  iostream::, 188
  マニュアルページ, 173
ild、コンパイル構成要素, 19
.il、ファイル名接尾辞, 11
imag、complex::, 219
include キーワード、テンプレートオプション ファイル, 86
include ファイル、検索順序, 273
includes のマニュアルページ, 173
include ディレクトリ、テンプレート定義ファイル, 83
include 文、オプションファイル, 86
indirect_array のマニュアルページ, 173
inline、コンパイル構成要素, 19
inner_product のマニュアルページ, 174
inplace_merge のマニュアルページ, 174
Input_Iterators のマニュアルページ, 168
inserter のマニュアルページ, 174
Insert_Iterators のマニュアルページ, 168
insert_iterator のマニュアルページ, 174
-instances=*a*、コンパイラオプション, 78 ~ 81, 38, 276
iomanip.h、iostream ヘッダーファイル, 189
ios_base のマニュアルページ, 174
iosfwd のマニュアルページ, 174
io_state、ios::, 192
iostream
  iostream::, 187
  新しいマルチスレッドインタフェース関数, 132 ~ 134
  エラービット, 192
  機能の拡張、マルチスレッド, 137
  共有版, 186
  構造, 187 ~ 188
  コンストラクタ, 188
  事前定義, 187
  出力, 189
  出力エラー, 191 ~ 192
  使用方法, 188
  シングルスレッドアプリケーション, 127
  従来の iostreams, 147
  入力, 193
  標準の iostreams, 147
  フラッシュ, 193
  ヘッダファイル, 188
  マニュアルページ, 185
  マルチスレッドで使用しても安全な、再入可能な関数, 126
  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, 131
  マルチスレッドでの制約, 127
  マルチスレッド用の新しいクラス階層, 132
  ライブラリ, 146, 151 ~ 152, 154
iostream.h、iostream ヘッダーファイル, 189, 134
iostreams
  make の使用, 26
  ファイルのアクセス, 282
iostream新旧形式の混合, 283
iostream ライブラリ, 185
ios のマニュアルページ, 174
ir2hf、コンパイル構成要素, 19
iropt、コンパイル構成要素, 19
isalnum のマニュアルページ, 174

isalpha のマニュアルページ, 174
iscntrl のマニュアルページ, 174
isdigit のマニュアルページ, 174
isgraph のマニュアルページ, 174
islower のマニュアルページ, 174
ISO C++ 標準
  準拠, 1
  一定義規約, 74
ISO C++ 標準、単一定義規則, 83
isprint のマニュアルページ, 174
ispunct のマニュアルページ, 174
isspace のマニュアルページ, 174
istream
  iostream::, 187
  マニュアルページ, 174
istreambuf_iterator のマニュアルページ, 174
istream_iterator のマニュアルページ, 174
istringstream のマニュアルページ, 175
istrstream
  iostream::, 188
  マニュアルページ, 175
isupper のマニュアルページ, 175
isxdigit のマニュアルページ, 175
_iterator_category のマニュアルページ, 169
Iterators のマニュアルページ, 168
iterator_traits のマニュアルページ, 175
iterator のマニュアルページ, 175
iter_swap のマニュアルページ, 175
-I-、コンパイラオプション, 273
-I、コンパイラオプション, 38, 84, 272
-i、コンパイラオプション, 32, 276
.i、ファイル名接尾辞, 10

K

.KEEP_STATE、<istream> 標準ライブラリ
  ヘッダーファイルとの使用, 26
-keeptmp、コンパイラオプション, 30, 277
-KPIC、コンパイラオプション, 29, 228, 277
-Kpic、コンパイラオプション, 29, 228, 277

L

-l, コンパイラオプション, 28
LD_LIBRARY_PATH 環境変数, 226
ld、コンパイル構成要素, 19
less_equal のマニュアルページ, 175
less のマニュアルページ, 175
lex、C++ ユーティリティ, 6
lexicographical_compare のマニュアルページ, 175
libC
  新しいマルチスレッドクラス, 131
  互換モード, 185, 189
  マルチスレッド環境での使用, 123
  マルチスレッドで使用しても安全なプログラムのコンパイルとリンク, 126
  ライブラリ, 146 ~ 147
libcomplex、complex参照
libcomplex、互換モード, 215
libCrun ライブラリ, 117, 118
libCstd ライブラリ、C++ 標準ライブラリ参照
libCstd、標準モード, 215
libc ライブラリ, 145
libdemangle ライブラリ, 146 ~ 148
libiostream
  標準モード, 185, 189
libiostream、iostream参照
-libmieee、コンパイラオプション, 279
-libmil、コンパイラオプション, 279
-libm、コンパイラオプション
  インラインテンプレート, 331
  最適化バージョン、, 331
libm ライブラリ, 145
-library=%all、コンパイラオプション, 34
-library、コンパイラオプション, 14, 32, 279 ~ 282
librwtool、Tools.h++を参照

libw ライブラリ, 145
limits のマニュアルページ, 175
limit、コマンド, 22
__LINE__、事前定義マクロ, 244
list のマニュアルページ, 175
locale のマニュアルページ, 175
log10、complex::, 219 ~ 221
log、complex::, 219 ~ 221
logical_and のマニュアルページ, 175
logical_not のマニュアルページ, 175
logical_or のマニュアルページ, 175
lower_bound のマニュアルページ, 175
-lpthread および POSIX, 278
-lthread
  -xnolibによる抑制, 156
  代わりに -mt を使用, 117, 126
-L、コンパイラオプション, 32, 277
-l、コンパイラオプション, 32, 278

M

make_heap のマニュアルページ, 175
make コマンド, 24 ~ 26
map のマニュアルページ, 175
mask_array のマニュアルページ, 175
math.h、complex ヘッダーファイル, 223
max_element のマニュアルページ, 176
max のマニュアルページ, 175
-mc、コンパイラオプション, 284
mem_fun1 のマニュアルページ, 176
mem_fun_ref1 のマニュアルページ, 176
mem_fun_ref のマニュアルページ, 176
mem_fun のマニュアルページ, 176
merge のマニュアルページ, 176
messages_byname のマニュアルページ, 176
messages のマニュアルページ, 176
-migration、コンパイラオプション, 30, 34, 38, 285
min_element のマニュアルページ, 176
minus のマニュアルページ, 176
min のマニュアルページ, 176
-misalign、コンパイラオプション, 14, 285 ~ 286
mismatch のマニュアルページ, 176
modulus のマニュアルページ, 176
money_get のマニュアルページ, 176
moneypunct_byname のマニュアルページ, 176
moneypunct のマニュアルページ, 176
money_put のマニュアルページ, 176
-mr コンパイラオプション, 286
-mt コンパイラオプション
  スレッドオプション, 39
  オプションの説明, 286 ~ 287
  コード生成オプション, 29
  コンパイルとリンク, 14
  ライブラリオプション, 32
  と libthread, 126
multimap のマニュアルページ, 176
multiplies のマニュアルページ, 176
multiset のマニュアルページ, 176
mutableキーワード、 認識, 253

N

namespace キーワード、認識, 255
-native、コンパイラオプション, 287
negate のマニュアルページ, 177
Negators のマニュアルページ, 168
new 列形式、認識, 255
next_permutation のマニュアルページ, 177
nocheck、フラグ, 87
-noex、コンパイラオプション, 287
-nofstore、コンパイラオプション, 31, 287 ~ 288
-nolibmil、コンパイラオプション, 288
-nolib、コンパイラオプション, 288
-noqueue、コンパイラオプション, 33, 288
norm、complex::, 219
-norunpath、コンパイラオプション, 32, 288

not1 のマニュアルページ, 177
not2 のマニュアルページ, 177
not_equal_to のマニュアルページ, 177
nth_element のマニュアルページ, 177
numeric_limits のマニュアルページ, 177
num_get のマニュアルページ, 177
numpunct_byname のマニュアルページ, 177
numpunct のマニュアルページ, 177
num_put のマニュアルページ, 177

O

-o *filename*、コンパイラオプション, 34, 289
ofstream
  iostream::, 188
  マニュアルページ, 177
-O*level*、コンパイラオプション, 289
Operators のマニュアルページ, 168
ostream
  iostream::, 187
  マニュアルページ, 177
ostreambuf_iterator のマニュアルページ, 177
ostream_iterator のマニュアルページ, 177
ostringstream のマニュアルページ, 177
ostrstream
  iostream::, 188
  マニュアルページ, 177
Output_Iterators のマニュアルページ, 168
overflow 関数、streambuf, 138
-O、コンパイラオプション, 289
.o ファイル
  オプション接尾辞, 11
  保護, 13

P

+p、コンパイラオプション, 290
pair のマニュアルページ, 177
partial_sort_copy のマニュアルページ, 178
partial_sort のマニュアルページ, 178
partial_sum のマニュアルページ, 178
partition のマニュアルページ, 178
PATH 環境変数、設定, xxviii
peek、istream::, 196
Pentium, 359
-pentium、コンパイラオプション, 291
permutation のマニュアルページ, 178
-pg、コンパイラオプション, 291
-PIC、コンパイラオプション, 291
-pic、コンパイラオプション, 291
plus のマニュアルページ, 178
pointer_to_binary_function のマニュアルページ, 178
pointer_to_unary_function のマニュアルページ, 178
polar、complex::, 217, 219
pop_heap のマニュアルページ, 178
POSIX および -lpthread, 278
pow、complex::, 219
#pragma キーワード, 373
Predicates のマニュアルページ, 168
prev_permutation のマニュアルページ, 178
priority_queue のマニュアルページ, 178
private、オブジェクトスレッド, 135
prof、C++ ユーティリティ, 6
Programming Language C++、標準の準拠, 1
-pta、コンパイラオプション, 292
ptclean コマンド, 78
pthread_cancel() 関数, 119
-pti、コンパイラオプション, 38, 84, 292
-pto、コンパイラオプション, 292
ptr_fun のマニュアルページ, 178
-ptr、コンパイラオプション, 34, 292
-ptv、コンパイラオプション, 293
push_heap のマニュアルページ, 178
-P、コンパイラオプション, 30, 34, 37, 290
-p、コンパイラオプション, 14, 37, 291

Q

-Qoption *phase option[,...option]*、コンパイラオプション, 30, 293 ~ 294
-qoption *phase option[,...option]*、コンパイラオプション, 294
-Qproduce *sourcetype*、コンパイラオプション, 34, 294
-qproduce *sourcetype*、コンパイラオプション, 295
-qp、コンパイラオプション, 294
queue のマニュアルページ, 178

R

Random_Access_Iterators のマニュアルページ, 168
random_shuffle のマニュアルページ, 178
raw_storage_iterator のマニュアルページ, 178
read、istream::, 196
-readme、コンパイラオプション, 30, 295
README ファイル, 2
real、complex::, 219
reinterpret_cast 演算子, 100, 101
remove_copy_if のマニュアルページ, 179
remove_copy のマニュアルページ, 178
remove_if のマニュアルページ, 179
remove のマニュアルページ, 178
replace_copy_if のマニュアルページ, 179
replace_copy のマニュアルページ, 179
replace_if のマニュアルページ, 179
replace のマニュアルページ, 179
return_temporary_buffer のマニュアルページ, 179
_reverse_bi_iterator のマニュアルページ, 169
reverse_copy のマニュアルページ, 179
reverse_iterator のマニュアルページ, 179
reverse のマニュアルページ, 179
RogueWave
  C++ 標準ライブラリ, 166
    Tools.h++も参照
rotate_copy のマニュアルページ, 179
rotate のマニュアルページ, 179
rttiキーワード、認識, 256
-R、コンパイラオプション, 28, 32, 295

S

-sbfast、コンパイラオプション, 296
-sb、コンパイラオプション, 296
search_n のマニュアルページ, 179
search のマニュアルページ, 179
Sequences のマニュアルページ, 168
set_difference のマニュアルページ, 179
set_intersection のマニュアルページ, 179
set_symmetric_difference のマニュアルページ, 179
set_terminate() 関数, 119
set_unexpected() 関数, 119
set_union のマニュアルページ, 179
set のマニュアルページ, 179
sh(1)、マニュアルページ, 22
sin、complex::, 219
sinh、complex::, 219 ~ 221
skip フラグ、iostream, 196
slice_array のマニュアルページ, 180
slice のマニュアルページ, 180
smanip_fill のマニュアルページ, 180
snamip のマニュアルページ, 180
Solaris オペレーティング環境ライブラリ, 145
Solaris プラットフォーム
  静的ライブラリの利用性, 238
sort_heap のマニュアルページ, 180
sort のマニュアルページ, 180
.so、ファイル名接尾辞, 11, 226
__sparc、事前定義マクロ, 245
sparc、事前定義マクロ, 245

__sparcv9、事前定義マクロ, 245
special、テンプレートコンパイルオプション, 77, 90
sqrt、complex::, 220
stable_partition のマニュアルページ, 180
stable_sort のマニュアルページ, 180
stack のマニュアルページ, 180
static_cast 演算子, 102
-staticlib、コンパイラオプション, 32, 150, 296 ~ 299
static、テンプレートクラスメンバーの特殊化, 92
__STDC__、事前定義マクロ, 51, 244
stdiostream.h、iostream ヘッダーファイル, 189
STL (標準テンプレートライブラリ)、構成要素, 165
stream.h、iostream ヘッダーファイル, 189
streambuf
  iostream::, 187
  新しい関数, 132
  公開仮想関数, 138
  マニュアルページ, 180
  ロック, 125
Stream_Iterators のマニュアルページ, 169
stream_locker
  マニュアルページ, 137
  マルチスレッドで使用しても安全なオブジェクトの同期処理, 131
stringbuf のマニュアルページ, 180
stringstream のマニュアルページ, 180
strstream
  iostream::, 188
  マニュアルページ, 180
strstream.h、iostream ヘッダーファイル, 189
strstreambuf
  マニュアルページ, 180
struct、名前のない宣言, 46
__sun、事前定義マクロ, 244
sun、事前定義マクロ, 244
__SUNPRO_CC_COMPAT=(4|5)、事前定義マクロ, 244
__SUNPRO_CC=0x510、事前定義マクロ, 244
SunWS_cache, 82
SunWS_config ディレクトリ, 85
__SVR4、事前定義マクロ, 244
swap -s、コマンド, 21
swap(1M)、マニュアルページ, 21
swap_ranges のマニュアルページ, 181
swap のマニュアルページ, 181
-S、コンパイラオプション, 296
-s、コンパイラオプション, 30, 34, 35, 296
.S、ファイル名接尾辞, 10
.s、ファイル名接尾辞, 10

T

tan、complex::, 220
tanh、complex::, 220
tcov、C++ ユーティリティ, 6
-temp=*dir*、コンパイラオプション, 30, 299
-template、コンパイラオプション, 83, 299 ~ 300
terminate() 関数, 119
thr_exit() 関数, 119
thr_keycreate、マニュアルページ, 135
time_get_byname のマニュアルページ, 181
time_get のマニュアルページ, 181
time_put_byname のマニュアルページ, 181
time_put のマニュアルページ, 181
-time、コンパイラオプション, 300
__TIME__、事前定義マクロ, 244
tolower のマニュアルページ, 181
Tools.h++
  コンパイラオプション, 154
  従来のおよび標準の iostreams, 147
  デバッグライブラリ, 146
  標準および互換モード, 147
  マニュアル, 147
toupper のマニュアルページ, 181

transform のマニュアルページ, 181

U

ube、コンパイル構成要素, 19
ulimit、コマンド, 22
__'uname-s'_'uname-r'、事前定義マクロ, 245
unary_function のマニュアルページ, 181
unary_negate のマニュアルページ, 181
unexpected() 関数, 119
uninitialized_copy のマニュアルページ, 181
uninitialized_fill_n のマニュアルページ, 181
uninitialized_fill のマニュアルページ, 181
unique_copy のマニュアルページ, 181
unique のマニュアルページ, 181
UNIX
  ツール, 6
__unix、事前定義マクロ, 245
unix、事前定義マクロ, 245
UNIX ツール, 6
-unroll=*n*、コンパイラオプション, 301
upper_bound のマニュアルページ, 181
use_facet のマニュアルページ, 181
-U、コンパイラオプション, 28, 37, 300

V

valarray のマニュアルページ, 181
-vdelx、コンパイラオプション, 34, 301
vector のマニュアルページ, 181
-verbose=no%template、テンプレートコンパイルオプション, 77
-verbose=template、テンプレートコンパイルオプション, 77
-verbose、コンパイラオプション, 302
-verbose、コマンド行オプション, 302, 15, 30
void *()、ios::, 192
-V、コンパイラオプション, 301
-v、コンパイラオプション, 301

W

+w、コンパイラオプション, 35
+w2、コンパイラオプション, 303
-w、コンパイラオプション, 35, 304
wcerr のマニュアルページ, 182
_WCHAR_T、UNIX 用の事前定義シンボル, 245
wcin のマニュアルページ, 182
wclog のマニュアルページ, 182
wcout のマニュアルページ, 182
wfilebuf のマニュアルページ, 182
wifstream のマニュアルページ, 182
wios のマニュアルページ, 182
wistream のマニュアルページ, 182
wistringstream のマニュアルページ, 182
wofstream のマニュアルページ, 182
wostream のマニュアルページ, 182
wostringstream のマニュアルページ, 183
write、ostream::, 193
ws、iostream マニピュレータ, 197
wstreambuf のマニュアルページ, 183
wstream のマニュアルページ, 182
wstringbuf のマニュアルページ, 183
wstring のマニュアルページ, 183
+w、コンパイラオプション, 77, 303

X

-xalias_level, compiler option, 305
-xarch=*isa*、コンパイラオプション, 14, 35, 308 ~ 313
-xar、コンパイラオプション, 32, 79, 226, 307
-xa、コンパイラオプション, 14, 37, 304 ~ 305
-xbuiltin、コンパイラオプション, 32, 313

-xcache=*c*、コンパイラオプション, 36, 315 ~ 316
-xcg89、コンパイラオプション, 14, 36, 316
-xcg92、コンパイラオプション, 14, 36, 316
-xcheck、コマンド行オプション, 30
-xchip=*c*、コンパイラオプション, 36, 317 ~ 319
-xcode=*a*、コンパイラオプション, 29, 319 ~ 320
-xcrossfile*[=n]*、コンパイラオプション, 321 ~ 322
-xF、コンパイラオプション, 36, 322
-xhelp=flags、コンパイラオプション, 30, 35, 38, 322
-xhelp=readme、コンパイラオプション, 35, 38, 323
-xia、コンパイラオプション, 32, 323
-xildoff、コンパイラオプション, 30, 324
-xildon、コンパイラオプション, 30, 324
-xinline=flst、コンパイラオプション, 36
-xinline、コンパイラオプション, 325
-xipo、コンパイラオプション, 36, 327
-xlang=l、コンパイラオプション, 32
-xlang、コンパイラオプション, 329
-xlibmieee、コンパイラオプション, 31, 33, 330
-xlibmil、コンパイラオプション, 33, 36, 331
-xlibmopt、コンパイラオプション, 33, 36, 331
-xlicinfo、コンパイラオプション, 33, 333
-xlic_lib、コンパイラオプション, 33, 332
-xM1、コンパイラオプション, 35, 37, 38, 333
-xM、コンパイラオプション, 333
-xMerge、コンパイラオプション, 29, 333
-Xm、コンパイラオプション, 333
-xM、コンパイラオプション, 35, 37, 38
-xnativeconnect、コンパイラオプション, 33
-xnolibmil、コンパイラオプション, 33, 36, 338
-xnolibmopt、コンパイラオプション, 33, 36, 338
-xnolib、コンパイラオプション, 151, 155 ~ 156, 33, 335 ~ 337
-xO*level*、コンパイラオプション, 36, 339 ~ 343
-xpg、コンパイラオプション, 14, 37, 343
-xprefetch*[=a[*、*a]]*、コンパイラオプション, 343 ~ 346
-xprefetch_level、コンパイラオプション, 36
-xprefetch、コンパイラオプション, 36
-xprofile=tcov、コンパイラオプション, 37
-xprofile、コンパイラオプション, 14, 347 ~ 350
-xregs=no%appl、コンパイラオプション, 229
-xregs、コンパイラオプション, 36, 350
-xsafe=mem、コンパイラオプション, 36, 39, 352
-xsbfast、コンパイラオプション, 30, 35, 353
-xsb、コンパイラオプション, 30, 35, 353
-xspace、コンパイラオプション, 36, 353
-xs、コンパイラオプション, 30, 352
-xtarget=*t*、コンパイラオプション, 14, 36, 353 ~ 360
-xtime、コンパイラオプション, 35, 360
-xtrigraphs、コンパイラオプション, 361, 31
-xunroll=*n*、コンパイラオプション, 361, 36
-xvector、コンパイラオプション, 362
-xwe、コンパイラオプション, 35, 363
X 挿入子、iostream, 189

Y

yacc、C++ ユーティリティ, 6

Z

-z *arg*、コンパイラオプション, 29, 35, 363
-ztext、コンパイラオプション, 363

あ

アクセスできるマニュアル, xxx

アセンブラ、コンパイル構成要素, 19
アセンブリ言語テンプレートのインライン展開, 19
値
  flush, 193
  挿入、cout, 190
  マニピュレータ, 189
値クラスの使用, 112
アナクロニズム、禁止する, 252
アプリケーション
  マルチスレッドアプリケーションのリンク, 117, 126
  マルチスレッドで使用しても安全, 123
  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, 139 ~ 141
アンダーフロー, 267
暗黙的インスタンス, 81

い

依存関係
  C++ 実行時ライブラリにおける、削除, 230
  共有ライブラリ, 228
一時オブジェクト, 109
インクリメンタルリンクエディタ、コンパイラ構成要素, 19
インスタンス化
  オプション, 78 ~ 81
  テンプレート関数, 62
  テンプレート関数メンバー、静的, 64
  テンプレート関数メンバー、明示的, 63
  テンプレートクラス, 62
インスタンス化、コンパイル時とリンク時, 84
インスタンスの状態、一貫性, 84
インスタンスメソッド
  静的, 80
  大域, 80
  テンプレート, 78
  半明示的, 81
  明示的, 80
インライン関数
  オプティマイザによる, 325
インライン関数の使用, 110
インラインテンプレート, 331

う

右辺値, 101

え

エラー
  チェック、マルチスレッドで使用しても安全, 127
エラー処理
  complex, 219 ~ 221
エラービット, 192
エラーメッセージ
  コンパイラのバージョンによる非互換性, 11
  リンカー, 16
  リンクの失敗, 13
エラーメッセージ、complex_error, 218
演算子
  complex, 221
  iostream, 189 ~ 193
  基本的な算術演算子, 218
  スコープ決定, 129
演算ライブラリ、複素数, 215 ~ 224

お

オーバーフロー, 267
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性能, 129, 131
オブジェクト、一時, 109
オブジェクトファイル
  再配置可能な, 228
  実行の順序, 28
オブジェクト、ライブラリ, 225
オプション
  アルファベット順リストにある個々のオプションも参照

キーワードファイルエントリ, 85
言語, 31
構文形式, 27, 29, 235
コード生成, 29
最適化, 35
サブプログラムのコンパイル, 15
出力, 34 ~ 35
処理の順序, 9, 28
スレッド, 39
説明、見出し, 236
ソース, 38
デバッグ, 30
展開コンパイル, 249
テンプレート, 38
テンプレートコンパイル, 79 ~ 84
テンプレート, 85
テンプレートコンパイル, 79
認識できない, 16
廃止, 34, 292
パフォーマンス, 34 ~ 37, 35
ファイル, 83 ~ 86, 91
浮動小数点, 31
プリプロセッサ, 37
プロファイル, 37
ライセンス, 33
ライブラリのリンク, 32
リファレンス, 38
オプション以外のもの、認識できない, 16
オプティマイザでのメモリー不足, 22
オブジェクト
stream_locker, 138
一時、存続期間, 255
共有オブジェクトの取り扱い方法, 135
共有オブジェクトの破棄, 138
大域的な共有, 134
破棄する順序, 255
ライブラリ内、リンク時, 225
オブジェクトスレッド、private, 135

か

ガーベージコレクション
ライブラリ, 148
外部インスタンス, 78
外部リンケージ, 79
各国語のサポート、アプリケーションの開発, 6
拡張機能, 43
定義, 1
非標準コードの許可, 253
拡張機能、定義, 1
角度、複素数, 216
数、複素数, 215
仮想メモリー、制限, 22 ~ 23
環境変数
LD_LIBRARY_PATH, 226
CCFLAGS, 24
SUNWS_CACHE_NAME, 82
SUNWS_CONFIG_NAME, 85
関数
streambuf の公開仮想関数, 138
オーバーライド, 43
オプティマイザによるインライン化, 325
静的、クラスフレンドとしての使用, 48
動的 (共有) ライブラリ, 227
マルチスレッドで使用しても安全な公開関数, 125
関数、__func__ での名前, 49
関数テンプレート, 57 ~ 58
使用, 58
宣言, 57
定義, 58
関数、動的 (共有) ライブラリ, 227
外部
インスタンス, 78
リンケージ, 79
ガベージコレクション
デバッグ、コンパイラオプション, 155
ライブラリ, 155

き

キーワード、オプションファイルエントリ, 85
記憶領域のサイズ, 370

規則、多重定義, 223
規則、単一定義, 83
キャッシュ
  オプティマイザを使用, 314
キャスト
  const と volatile, 100
  reinterpret, 101
  静的, 102
  動的, 102
    void* へキャスト, 103
    下位へキャスト, 102
    上位へキャスト, 102
キャッシュディレクトリ、テンプレート, 11
共有オブジェクト, 135, 138
局所スコープ規則、使用する、使用しない, 253
共役複素数, 216
共有ライブラリ
  C プログラムからのアクセス, 231
  構築, 268
  名前の割り当て, 272
  リンクの禁止, 244
  例外を含む, 228
  構築、例外付き, 97
極座標、複素数, 216

く

空白
  抽出子, 196
  読み飛ばし, 196
区間演算ライブラリのリンク, 323
クラス、間接的に渡す, 112
クラスインスタンス、名前のない, 47
クラステンプレート, 59 ~ 62
  使用, 61
  静的データメンバー, 61
  宣言, 59
  定義, 59
  メンバー、定義, 60

け

警告
  アナクロニズム, 252
  移植性がないコード, 303
  移植性を損なう技術的な違反, 303
  移植性のないコード, 303
  効率が悪いコード, 303
  コードの移植性, 251
  認識できない引数, 16
  非効率なコード, 303
  問題のある ARM 言語の構造, 254
  抑制, 304
言語
  オプション, 31
  各国語のサポート, 6
検索パス
  定義, 84
  動的ライブラリ, 151
  標準ヘッダーの実装, 160 ~ 161
  include ファイル、定義済みの, 272
  ソースオプション, 38
  テンプレートオプション, 38

こ

公開関数、マルチスレッドで使用しても安全, 125
構文, 235
  CC コマンド, 8
  オプション, 29, 235
コード
  オプティマイザ, 19
  生成オプション, 29
コードオプティマイザ、コンパイル構成要素, 19
コード生成
  インライン機能、アセンブラ、コンパイル構成要素, 19
互換モード
  libC, 185, 189
  libcomplex, 215
国際化、実装, 6
コマンド行

オプション、診断, 16
認識できるファイル名接尾辞, 10
混合モード、複素数演算ライブラリ, 223
混合言語のリンク, 329
コンストラクタ
complex, 217
complex クラス, 216
iostream, 188
静的, 228
コンストラクタ、静的, 227
コンパイラ
構成要素, 19
構成要素を呼び出す順序, 16 ~ 19
新機能, 3
診断, 15 ~ 16
バージョン、非互換性, 11
ube_ipa、コンパイル構成要素, 19
コンパイル単位、複数のソースファイル, 10 ~ 11
コンパイルとリンク, 13 ~ 14
コンパイル、メモリー条件, 21 ~ 23
コンパイル、アクセス, xxix
互換モード
Tools.h++, 147

さ

サイズ、記憶領域の, 370
最適化
オプションの, 35
数学ライブラリ, 331
対象となるハードウェア, 353
レベル, 339
先読み、入力の, 196
先読み命令、有効にする, 343
サブプログラム、コンパイルオプション, 15
左辺値, 101
三角関数、複素数演算ライブラリ, 220

し

シェルプロンプト, xxviii
シェル、仮想メモリーの制限, 22
識別子, 253
シグナルハンドラ
とマルチスレッド, 118
と例外, 93
実際のメモリー、表示, 23
実数、複素数, 215
自動読み込み、dbx 用に無効化, 352
終了関数, 368
出力, 185
cout, 190
string, 191
エラー処理, 191
バイナリ, 193
文字列, 191
出力オプション, 34 ~ 35
書体と記号について, xxvi
初期化関数, 366, 367
書体表記, xxvi
処理、順序、オプションの, 9
指令、C++, 365
シンボル、実行可能ファイル, 296
事前定義マクロ, 244
実行時エラーメッセージ, 94
実行時関数
事前定義済み, 95
実数、複素数, 218
従来のリンクエディタ、コンパイル構成要素, 19

す

数、複素数, 218
数学関数、複素数演算ライブラリ, 220
数学ライブラリ、最適化したバージョン, 331
数学関数、複素数演算ライブラリ<>, 218
スコープ決定演算子、unsafe_ クラス, 129
スレッドオプション, 39
スワップ領域, 21 ~ 23

せ

静的
  関数、参照, 74
  非ローカル静的オブジェクトの初期設定子, 254
  変数、参照, 74
静的 (アーカイブ) ライブラリ, 225
静的インスタンス, 78 ~ 80
静的データ、マルチスレッドアプリケーション, 134
静的リンク
  コンパイラが提供するライブラリ, 150, 296, 299
静的リンケージ
  テンプレートインスタンス, 79
性能
  オプション, 35
  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, 129, 131
制約、マルチスレッドで使用しても安全な
  `iostream`, 127
整列
  厳密な, 370
  デフォルト, 370
絶対値、複素数, 216
接尾辞
  コマンド行、ファイル名, 10
  メークファイル, 24 ~ 26
  ライブラリ, 226
  接尾辞のないファイル, 160
  テンプレート定義ファイル, 86

そ

相互排他領域、定義, 137
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, 130, 138
挿入
  演算子, 189 ~ 191
ソースオプション, 38
ソースコンパイラオプション, 38
ソースファイル
  テンプレート定義, 86
  位置規約, 83
  位置定義, 86 ~ 90
  実行の順序, 28

た

大域
  データ、マルチスレッドアプリケーション, 134
  マルチスレッドアプリケーションでの共有オブジェクト, 134
大域インスタンス, 78 ~ 80
大域リンケージ, 78 ~ 81
多重定義、規則 9, 223

ち

中間言語トランスレータ、コンパイル構成要素, 19
抽出
  演算子, 194
抽出子
  `char*`, 194 ~ 195
  空白, 196
  ユーザー定義の `iostream`, 194

て

定義, 59
定義、テンプレートの検索, 83
ディレクトリ、名前の変更, 85
データ型、複素数, 215 ~ 216
デストラクタ、静的, 227
デバッグ
  オプション, 29, 30
  コンパイル時のインスタンス化, 84
  修正継続機能, 79
  プログラムの準備, 15
デフォルト演算子の使用, 111
テンプレート

インスタンス、コンパイル単位間で共有, 81
インスタンスの再コンパイルの制御, 85
インスタンスメソッド, 78, 84
インライン, 331
オプション, 38
オプションファイルの共有, 86
キャッシュディレクトリ, 11
共有オプションファイル, 86
コマンド, 78
コンパイル, 79
冗長コンパイル, 77
静的オブジェクト、参照, 74
ソースファイル, 83, 86 ~ 90
定義-検索オプション, 85
定義分離型と定義取り込み型, 83
特殊化, 65
使用, 67
宣言, 65
定義, 66
特殊化エントリ, 90 ~ 77
特殊化の制御, 85
標準テンプレートライブラリ (STL), 165
明示的なインスタンス化を制御, 85
リポジトリ, 82
リンク, 15
テンプレート、入れ子になった, 64
テンプレートクラス、特殊化, 91
テンプレートコンパイル, 79 ~ 84
テンプレートのインスタンス化, 62
暗黙的, 62
関数, 62
全クラス, 62
明示的, 63
テンプレートの特殊化, 65 ~ 67
テンプレートの問題, 68
テンプレート関数のフレンド宣言, 70
テンプレートで修飾名を使う
定義, 73
引数としての局所型, 70
非局所型名前の解決とインスタンス化, 68
テンプレートパラメータ、デフォルト, 65
テンプレートオプションファイル, 85
テンプレート定義
検索パス, 84
取り込み, 54
別の、ファイル, 83
テンプレートの問題
静的オブジェクト、参照, 74
テンプレートのリンクの前処理、コンパイル構成要素, 19
デバッグ
オブジェクトファイルなし, 352

と

動的 (共有) ライブラリ, 225 ~ 228
トークンの代替スペル, 252
トラップモード, 266
トリグラフシーケンスの認識, 361

な

内部手続きの最適化, 327
内部処理アナライザ、コンパイル構成要素, 19
名前のないクラスインスタンス、受け渡し, 47
名前、変更、ディレクトリの, 85

に

入出力、complex, 221, 185
入力
iostream, 193
先読み, 196
バイナリ, 196

は

ハードウェアのアーキテクチャ, 354
配置、テンプレートインスタンス, 78
バイナリ入力、読み取り, 196
パフォーマンスオプション, 34 ~ 37
半明示的インスタンス, 78, 81

ひ

非互換性、コンパイラのバージョン, 11
左シフト演算子
  complex, 221
  iostream, 189
非標準機能
  定義, 1
  非標準コードの許可, 253
, 43
非標準機能、定義, 1
標準、準拠, 1
標準ストリーム、iostream.h, 134
標準テンプレートライブラリ (STL), 165
標準 iostream クラス, 185
標準エラー、iostreams, 187
標準出力、iostreams, 187
標準入力、iostreams, 187
標準ヘッダー
  置き換え, 162
  実装, 160
標準モード
  iostream, 185
  libCstd, 215
  libiostream, 185, 189

ふ

ファイル
  C 標準ヘッダーファイル, 160
  テンプレートオプション, 85
  標準ライブラリ, 160
  複数のソースファイルの使用, 11
  オブジェクト, 13, 28, 228
  オプション, 83 ~ 86, 91
  実行可能プログラム, 13
  複数のソース, 10 ~ 11
ファイル名
  接尾辞, 10
  テンプレート定義ファイル, 83
  標準ライブラリ, 26
不完全な, 59
複数のソースファイルの使用, 11
浮動小数点
  オプション, 31
  精度モード, 262
  無効な, 266
プラグマ
  #pragma unknown_control_flow 指令, 370
プリプロセッサ
  オプション, 37
  マクロの定義, 243
プログラム、基本的な構築手順, 7 ~ 8
プロセッサ、対象となる、指定, 354
プロファイルオプション, 37, 347
フロントエンド、コンパイルコンポーネント, 16
ブール型とリテラル、許可する, 252
プログラム
  基本的な構築手順, 7 ~ 9
  マルチスレッドプログラムの構築, 117

へ

ヘッダーファイル
  C 標準, 160
  iostream, 134
  言語への対応, 51
  作成, 51
  標準ライブラリ, 158, 166 ~ 167
  べき, 53
  complex, 216, 223
  iostream, 189
ヘッダーファイル、不要なインクルード, 370
別名、定義, 24
変数引数リスト, 20
  #define, 20
べき, 51
別名、コマンドの簡略化, 23

ま

マクロ

マクロの一覧も参照, 244
マクロ、事前定義, 244
マニュアル索引, xxx
マニュアルページ、アクセス, xxviii
マニュアル、アクセス, xxx
マニュアルページ
C++ 標準ライブラリ, 168, 183
complex, 224
iostream, 185
sh(1), 22
swap(1M), 21
アクセス, 148
表示, 2
マニュアルへのアクセス, xxxii
マルチスレッド
アプリケーション, 118
コンパイル, 118
例外処理, 119
マルチスレッドで使用しても安全
アプリケーション, 123
オブジェクト, 123
クラス、派生時の注意点, 138
公開関数, 125
性能のオーバーヘッド, 129, 131
ライブラリ, 123
マニュアルページ, 180

み

右シフト演算子
complex, 221

め

明示的インスタンス, 78 ~ 80
メモリー条件, 21 ~ 23
メンバー変数のキャッシュ, 114

も

文字列
iostream::, 191

ゆ

ユーザー定義の型
iostream, 190
ユーザー定義型
マルチスレッドで使用しても安全, 128 ~ 129
優先順位の問題, 190

よ

読み取り専用メモリーでのconststrings, 252
読み取り専用メモリーでのリテラル文字
列、, 252

ら

ライセンス
オプション, 33
条件, 3
ライブラリ
C インタフェース, 145
C++ 標準, 165
C++ コンパイラ、提供, 146
Sun Performance Library、リンク, 281, 332
共有, 244
共有ライブラリへの名前の割り当て, 272
区間演算, 323
最適化した、数学, 331
実行の順序, 28
従来型の iostream, 185 ~ 214
使用, 145 ~ 163
静的, 238
接尾辞, 225
複素数演算, 215 ~ 224
理解, 225 ~ 226
リンクオプション, 32
ライブラリ、構築
C API による, 230
公開用, 229

静的 (アーカイブ), 225 ~ 227
動的 (共有), 225 ~ 228
非公開用, 229
リンクオプション, 32, 268
例外との共有, 228

り
リファレンスオプション, 38
リンカー, 352
リンク
complexライブラリ, 153 ~ 154
-mt オプション, 126
コンパイルとの整合性, 14 ~ 15
コンパイルとの分離, 14
システムライブラリとの、無効化, 335
静的 (アーカイブ) ライブラリ, 150
速度の向上, 352
テンプレートインスタンスメソッド, 78
動的 (共有) ライブラリ, 225 ~ 226
マルチスレッドで使用しても安全な libC ライブラリ, 126
ライブラリオプション, 32

れ
例外
トラップ, 267
標準クラス, 95
標準ヘッダー, 95
longjmp と, 96
setjmp と, 96
関数、オーバーライド, 43
禁止する, 253
シグナルハンドラと, 96
とマルチスレッド, 119
無効化, 94
例外のある共有ライブラリの構築, 97
連続、入出力操作の「マルチスレッドで使用しても安全」な実行, 135

ろ
ロック
streambuf, 125
オブジェクト, 135 ~ 137
相互排他ロック, 130, 138

わ
ワークステーション、メモリー条件, 23

ん
ファイル
「ソースファイル」も参照
ロック
stream_locker も参照