Oracle® Solaris Studio 12.2: C++ ユーザーズガイド

Part No: 821–2495
2010 年 9 月

ORACLE®

110412@25097

目次

はじめに 19

パート I **C++ コンパイラ** 25

1 C++ コンパイラの紹介 27
1.1 Solaris Studio 12.2 C++ 5.11 コンパイラの新機能 27
1.2 x86 の特記事項 28
1.3 64 ビットプラットフォーム用のコンパイル 29
1.4 バイナリの互換性の妥当性検査 29
1.5 準拠規格 30
1.6 C++ README ファイル 30
1.7 マニュアルページ 31
1.8 各国語のサポート 31

2 C++ コンパイラの使用方法 33
2.1 コンパイル方法の概要 33
2.2 コンパイラの起動 35
2.2.1 コマンド構文 35
2.2.2 ファイル名に関する規則 35
2.2.3 複数のソースファイルの使用 36
2.3 バージョンが異なるコンパイラでのコンパイル 37
2.4 コンパイルとリンク 37
2.4.1 コンパイルとリンクの流れ 37
2.4.2 コンパイルとリンクの分離 38
2.4.3 コンパイルとリンクの整合性 38
2.4.4 64 ビットメモリーモデル用のコンパイル 39
2.4.5 コンパイラのコマンド行診断 39

2.4.6 コンパイラの構成 40
2.5 指示および名前の前処理 41
2.5.1 プラグマ 41
2.5.2 可変数の引数をとるマクロ 41
2.5.3 事前に定義されている名前 42
2.5.4 警告とエラー 42
2.6 メモリー条件 43
2.6.1 スワップ領域のサイズ 43
2.6.2 スワップ領域の増加 43
2.6.3 仮想メモリーの制御 44
2.6.4 メモリー条件 45
2.7 C++ オブジェクトに対する `strip` コマンドの使用 45
2.8 コマンドの簡略化 45
2.8.1 C シェルでの別名の使用 45
2.8.2 `CCFLAGS` によるコンパイルオプションの指定 45
2.8.3 `make` の使用 46

3 C++ コンパイラオプションの使い方 47
3.1 構文の概要 47
3.2 一般的な注意事項 48
3.3 機能別に見たオプションの要約 48
3.3.1 コード生成オプション 48
3.3.2 コンパイル時パフォーマンスオプション 49
3.3.3 コンパイル時とリンク時のオプション 50
3.3.4 デバッグオプション 51
3.3.5 浮動小数点オプション 52
3.3.6 言語オプション 53
3.3.7 ライブラリオプション 53
3.3.8 廃止オプション 55
3.3.9 出力オプション 56
3.3.10 実行時パフォーマンスオプション 57
3.3.11 プリプロセッサオプション 59
3.3.12 プロファイルオプション 59
3.3.13 リファレンスオプション 60
3.3.14 ソースオプション 60

3.3.15 テンプレートオプション 60
3.3.16 スレッドオプション 61

パート II C++ プログラムの作成 63

4 言語拡張 65
4.1 リンカースコープ 65
4.1.1 Microsoft Windows との互換性 67
4.2 スレッドローカルな記憶装置 67
4.3 例外の制限の少ない仮想関数による置き換え 68
4.4 enum の型と変数の前方宣言の実行 68
4.5 不完全な enum 型の使用 69
4.6 enum 名のスコープ修飾子としての使用 69
4.7 名前のない struct 宣言の使用 70
4.8 名前のないクラスインスタンスのアドレスの受け渡し 71
4.9 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 71
4.10 事前定義済み __func__ シンボルの関数名としての使用 72
4.11 サポートされる属性 72
4.11.1 __packed__ 73

5 プログラムの編成 75
5.1 ヘッダーファイル 75
5.1.1 言語に対応したヘッダーファイル 75
5.1.2 べき等ヘッダーファイル 76
5.2 テンプレート定義 77
5.2.1 テンプレート定義の取り込み 77
5.2.2 テンプレート定義の分離 78

6 テンプレートの作成と使用 81
6.1 関数テンプレート 81
6.1.1 関数テンプレートの宣言 81
6.1.2 関数テンプレートの定義 82
6.1.3 関数テンプレートの使用 82
6.2 クラステンプレート 82

6.2.1 クラステンプレートの宣言 82
6.2.2 クラステンプレートの定義 83
6.2.3 クラステンプレートメンバーの定義 83
6.2.4 クラステンプレートの使用 84
6.3 テンプレートのインスタンス化 85
6.3.1 テンプレートの暗黙的インタンス化 85
6.3.2 テンプレートの明示的インスタンス化 85
6.4 テンプレートの編成 86
6.5 デフォルトのテンプレートパラメータ 87
6.6 テンプレートの特殊化 87
6.6.1 テンプレートの特殊化宣言 87
6.6.2 テンプレートの特殊化定義 88
6.6.3 テンプレートの特殊化の使用とインスタンス化 88
6.6.4 部分特殊化 88
6.7 テンプレートの問題 89
6.7.1 非局所型名前の解決とインスタンス化 89
6.7.2 テンプレート引数としての局所型 90
6.7.3 テンプレート関数のフレンド宣言 90
6.7.4 テンプレート定義内での修飾名の使用 92
6.7.5 テンプレート名の入れ子 92
6.7.6 静的変数や静的関数の参照 93
6.7.7 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 93

7 テンプレートのコンパイル 97
7.1 冗長コンパイル 97
7.2 リポジトリの管理 97
7.2.1 生成されるインスタンス 98
7.2.2 全クラスインスタンス化 98
7.2.3 コンパイル時のインスタンス化 98
7.2.4 テンプレートインスタンスの配置とリンケージ 99
7.3 外部インスタンス 99
7.3.1 キャッシュの衝突 100
7.3.2 静的インスタンス 101
7.3.3 大域インスタンス 102

7.3.4 明示的インスタンス 102
7.3.5 半明示的インスタンス 102
7.4 テンプレートリポジトリ 103
7.4.1 リポジトリの構造 103
7.4.2 テンプレートリポジトリへの書き込み 103
7.4.3 複数のテンプレートリポジトリからの読み取り 104
7.4.4 テンプレートリポジトリの共有 104
7.4.5 `-instances=extern` によるテンプレートインスタンスの自動一貫性 104
7.5 テンプレート定義の検索 105
7.5.1 ソースファイルの位置規約 105
7.5.2 定義検索パス 105
7.5.3 問題がある検索の回避 106

8 例外処理 107
8.1 同期例外と非同期例外 107
8.2 実行時エラーの指定 107
8.3 例外の無効化 108
8.4 実行時関数と事前定義済み例外の使用 108
8.5 シグナルや `Setjmp/Longjmp` と例外との併用 109
8.6 例外のある共有ライブラリの構築 110

9 プログラムパフォーマンスの改善 111
9.1 一時オブジェクトの回避 111
9.2 インライン関数の使用 112
9.3 デフォルト演算子の使用 113
9.4 値クラスの使用 113
9.4.1 クラスを直接渡す 114
9.4.2 各種のプロセッサでクラスを直接渡す 114
9.5 メンバー変数のキャッシュ 115

10 マルチスレッドプログラムの構築 117
10.1 マルチスレッドプログラムの構築 117
10.1.1 マルチスレッドコンパイルの確認 117
10.1.2 C++ サポートライブラリの使用 118

10.2 マルチスレッドプログラムでの例外の使用 118
10.2.1 スレッドの取り消し 118
10.3 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
10.4 マルチスレッド環境での従来の iostream の使用 121
10.4.1 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 122
10.4.2 iostream ライブラリのインタフェースの変更 128
10.4.3 大域データと静的データ 130
10.4.4 連続実行 130
10.4.5 オブジェクトのロック 131
10.4.6 マルチスレッドで使用しても安全なクラス 133
10.4.7 オブジェクトの破棄 134
10.4.8 アプリケーションの例 134
10.5 メモリーバリアー組み込み関数 136

パート III ライブラリ 139

11 ライブラリの使用 141
11.1 C ライブラリ 141
11.2 C++ コンパイラ付属のライブラリ 141
11.2.1 C++ ライブラリの説明 142
11.2.2 C++ ライブラリのマニュアルページへのアクセス 144
11.2.3 デフォルトの C++ ライブラリ 144
11.3 関連するライブラリオプション 144
11.4 クラスライブラリの使用 146
11.4.1 iostream ライブラリ 146
11.4.2 complex ライブラリ 147
11.4.3 C++ ライブラリのリンク 148
11.5 標準ライブラリの静的リンク 149
11.6 共有ライブラリの使用 150
11.7 C++ 標準ライブラリの置き換え 151
11.7.1 置き換え可能な対象 151
11.7.2 置き換え不可能な対象 152
11.7.3 代替ライブラリのインストール 152
11.7.4 代替ライブラリの使用 152
11.7.5 標準ヘッダーの実装 153

12 C++ 標準ライブラリの使用 157
12.1 C++ 標準ライブラリのヘッダーファイル 158
12.2 C++ 標準ライブラリのマニュアルページ 159
12.3 STLport 173
12.3.1 再配布とサポートされる STLport ライブラリ 173
12.4 Apache stdcxx 標準ライブラリ 174

13 従来の iostream ライブラリの使用 175
13.1 定義済みの iostream 175
13.2 iostream 操作の基本構造 176
13.3 従来の iostream ライブラリの使用 177
13.3.1 iostream を使用した出力 178
13.3.2 iostream を使用した入力 180
13.3.3 ユーザー定義の抽出演算子 181
13.3.4 char* の抽出子 181
13.3.5 1 文字の読み込み 182
13.3.6 バイナリ入力 182
13.3.7 入力データの先読み 183
13.3.8 空白の抽出 183
13.3.9 入力エラーの処理 183
13.3.10 iostream と stdio の併用 184
13.4 iostream の作成 184
13.4.1 クラス fstream を使用したファイル操作 184
13.5 iostream の代入 187
13.6 フォーマットの制御 187
13.7 マニピュレータ 188
13.7.1 引数なしのマニピュレータの使用法 189
13.7.2 引数付きのマニピュレータの使用法 190
13.8 ストリーム: 配列用の iostream 192
13.9 stdiobuf: 標準入出力ファイル用の iostream 192
13.10 streambuf 192
13.10.1 streambuf の機能 192
13.10.2 streambuf の使用 193
13.11 iostream に関するマニュアルページ 193
13.12 iostream の用語 195

14 複素数演算ライブラリの使用 199
14.1 複素数ライブラリ 199
14.1.1 複素数ライブラリの使用方法 200
14.2 `complex` 型 200
14.2.1 `complex` クラスのコンストラクタ 200
14.2.2 算術演算子 201
14.3 数学関数 202
14.4 エラー処理 203
14.5 入出力 204
14.6 混合演算 205
14.7 効率 206
14.8 複素数のマニュアルページ 206

15 ライブラリの構築 207
15.1 ライブラリとは 207
15.2 静的 (アーカイブ) ライブラリの構築 208
15.3 動的 (共有) ライブラリの構築 209
15.4 例外を含む共有ライブラリの構築 210
15.5 非公開ライブラリの構築 210
15.6 公開ライブラリの構築 211
15.7 C API を持つライブラリの構築 211
15.8 `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 212

パート **IV** 付録 213

A **C++** コンパイラオプション 215
A.1 オプション情報の構成 215
A.2 オプションの一覧 216
A.2.1 `-386` 217
A.2.2 `-486` 217
A.2.3 `-B`*binding* 217
A.2.4 `-c` 219
A.2.5 `-cg{89|92}` 219
A.2.6 `–compat[={ 4|5|g}]` 219

A.2.7 +d 221
A.2.8 -D*name*[=*def*] 222
A.2.9 -d{y| n} 222
A.2.10 -dalign 223
A.2.11 -dryrun 224
A.2.12 -E 224
A.2.13 +e{0|1} 225
A.2.14 -erroff[=*t*] 226
A.2.15 -errtags[=*a*] 227
A.2.16 -errwarn[=*t*] 227
A.2.17 -fast 228
A.2.18 -features=*a*[,*a*...] 231
A.2.19 -filt[=*filter*[,*filter*...]] 235
A.2.20 -flags 237
A.2.21 -fma[={none| fused}] 238
A.2.22 -fnonstd 238
A.2.23 -fns[={yes| no}] 238
A.2.24 -fprecision=*p* 240
A.2.25 -fround=*r* 241
A.2.26 –fsimple[=*n*] 242
A.2.27 -fstore 243
A.2.28 -ftrap=*t*[,*t*...] 244
A.2.29 -G 245
A.2.30 -g 246
A.2.31 -g0 248
A.2.32 -H 248
A.2.33 -h[ ]*name* 248
A.2.34 -help 249
A.2.35 -I*pathname* 249
A.2.36 -I- 250
A.2.37 -i 252
A.2.38 -include *filename;* 252
A.2.39 -inline 253
A.2.40 -instances=*a* 253
A.2.41 -instlib=*filename* 254
A.2.42 -KPIC 255

A.2.43 -Kpic 256
A.2.44 -keeptmp 256
A.2.45 -L*path* 256
A.2.46 -l*lib* 256
A.2.47 -libmieee 257
A.2.48 -libmil 257
A.2.49 -library=*l*[,*l*...] 257
A.2.50 -m32|-m64 262
A.2.51 -mc 263
A.2.52 -migration 263
A.2.53 -misalign 263
A.2.54 -mr[, *string*] 264
A.2.55 -mt[={yes |no}] 264
A.2.56 -native 265
A.2.57 -noex 265
A.2.58 -nofstore 265
A.2.59 -nolib 265
A.2.60 -nolibmil 266
A.2.61 -noqueue 266
A.2.62 -norunpath 266
A.2.63 -O 266
A.2.64 -O*level* 267
A.2.65 -o *filename* 267
A.2.66 +p 267
A.2.67 -P 268
A.2.68 -p 268
A.2.69 -pentium 268
A.2.70 -pg 268
A.2.71 -PIC 268
A.2.72 -pic 268
A.2.73 -pta 269
A.2.74 -pti*path* 269
A.2.75 -pto 269
A.2.76 –ptr 269
A.2.77 -ptv 269
A.2.78 -Qoption *phase option*[,*option*...] 270

A.2.79 -qoption *phase option* 271
A.2.80 -qp 271
A.2.81 -Qproduce *sourcetype* 271
A.2.82 -qproduce *sourcetype* 271
A.2.83 -R*pathname*[:*pathname*...] 272
A.2.84 -readme 272
A.2.85 -S 272
A.2.86 -s 272
A.2.87 -sb 273
A.2.88 -sbfast 273
A.2.89 -staticlib=*l*[,*l*...] 273
A.2.90 -sync_stdio=[yes| no] 275
A.2.91 -temp=*path* 276
A.2.92 -template=*opt*[,*opt*...] 276
A.2.93 -time 278
A.2.94 -traceback[={ %none|common|*signals_list*}] 278
A.2.95 -U*name* 279
A.2.96 -unroll=*n* 279
A.2.97 -V 279
A.2.98 -v 280
A.2.99 -vdelx 280
A.2.100 -verbose=*v*[,*v*...] 280
A.2.101 +w 281
A.2.102 +w2 282
A.2.103 -w 282
A.2.104 -Xm 282
A.2.105 -xaddr32 282
A.2.106 -xalias_level[=*n*] 283
A.2.107 -xannotate[=yes| no] 285
A.2.108 -xar 286
A.2.109 -xarch=*isa* 287
A.2.110 -xautopar 292
A.2.111 -xbinopt={prepare| off} 292
A.2.112 -xbuiltin[={ %all|%none}] 293
A.2.113 -xcache=*c* 294
A.2.114 -xcg[89|92] 295

A.2.115 -xchar[= *o*] 296
A.2.116 -xcheck[= *i*] 297
A.2.117 -xchip=*c* 298
A.2.118 -xcode=*a* 300
A.2.119 -xcrossfile[= *n*] 302
A.2.120 -xdebugformat=[stabs|dwarf] 302
A.2.121 -xdepend=[yes| no] 303
A.2.122 -xdumpmacros[= *value*[,*value...*]] 304
A.2.123 -xe 307
A.2.124 -xF[=*v*[, *v*...]] 307
A.2.125 -xhelp=flags 308
A.2.126 -xhelp=readme 308
A.2.127 -xhwcprof 309
A.2.128 -xia 309
A.2.129 -xinline[=*func_spec*[,*func_spec...*]] 310
A.2.130 -xinstrument=[no%]datarace 312
A.2.131 -xipo[={0|1|2}] 313
A.2.132 -xipo_archive=[*a*] 315
A.2.133 -xjobs=*n* 316
A.2.134 -xkeepframe[=[%all,%none,*name*,no% *name*]] 317
A.2.135 -xlang=*language*[,*language*] 317
A.2.136 -xldscope={*v*} 319
A.2.137 -xlibmieee 320
A.2.138 -xlibmil 320
A.2.139 -xlibmopt 321
A.2.140 -xlic_lib=sunperf 321
A.2.141 -xlicinfo 321
A.2.142 -xlinkopt[= レベル] 322
A.2.143 -xloopinfo 323
A.2.144 -xM 323
A.2.145 -xM1 324
A.2.146 -xMD 325
A.2.147 -xMF 325
A.2.148 -xMMD 325
A.2.149 -Merge 325
A.2.150 -xmaxopt[=*v*] 326

A.2.151 -xmemalign=*ab* 326
A.2.152 -xmodel=[*a*] 328
A.2.153 -xnolib 329
A.2.154 -xnolibmil 330
A.2.155 -xnolibmopt 330
A.2.156 -xnorunpath 331
A.2.157 -xO*level* 331
A.2.158 -xopenmp[= *i*] 334
A.2.159 -xpagesize=*n* 336
A.2.160 -xpagesize_heap=*n* 337
A.2.161 -xpagesize_stack=*n* 337
A.2.162 -xpch=*v* 338
A.2.163 -xpchstop=*file* 341
A.2.164 -xpec[={yes|no}] 342
A.2.165 -xpg 342
A.2.166 -xport64[=(*v*)] 343
A.2.167 -xprefetch[=*a*[,*a*...]] 346
A.2.168 -xprefetch_auto_type=*a* 349
A.2.169 -xprefetch_level[=*i*] 349
A.2.170 –xprofile=*p* 350
A.2.171 -xprofile_ircache[=*path*] 353
A.2.172 -xprofile_pathmap 354
A.2.173 -xreduction 354
A.2.174 -xregs=*r*[,*r*...] 355
A.2.175 -xrestrict[=*f*] 357
A.2.176 -xs 359
A.2.177 -xsafe=mem 359
A.2.178 -xsb 360
A.2.179 -xsbfast 360
A.2.180 -xspace 360
A.2.181 -xtarget=*t* 360
A.2.182 -xthreadvar[= *o*] 364
A.2.183 -xtime 365
A.2.184 -xtrigraphs[={ yes|no}] 365
A.2.185 -xunroll=*n* 366
A.2.186 -xustr={ascii_utf16_ushort|no} 367

A.2.187 -xvector[=*a*] 368
A.2.188 -xvis[={yes|no}] 369
A.2.189 - xvpara 369
A.2.190 -xwe 370
A.2.191 -Y*c*,*path* 370
A.2.192 -z[ ]*arg* 371

B プラグマ 373
B.1 プラグマの書式 373
B.1.1 プラグマの引数としての多重定義関数 373
B.2 プラグマの詳細 374
B.2.1 #pragma align 374
B.2.2 #pragma does_not_read_global_data 375
B.2.3 #pragma does_not_return 375
B.2.4 #pragma does_not_write_global_data 376
B.2.5 #pragma dumpmacro s 376
B.2.6 #pragma end_dumpmacros 377
B.2.7 #pragma error_messages 378
B.2.8 #pragma fini 378
B.2.9 #pragma hdrstop 378
B.2.10 #pragma ident 379
B.2.11 #pragma init 379
B.2.12 #pragma must_have_frame 379
B.2.13 #pragma no_side_effect 380
B.2.14 #pragma opt 381
B.2.15 #pragma pack(*n*) 381
B.2.16 #pragma rarely_called 382
B.2.17 #pragma returns_new_memory 383
B.2.18 #pragma unknown_control_flow 383
B.2.19 #pragma weak 384

用語集 387

索引 393

例目次

例 6–1 テンプレート引数としての局所型の問題の例 90
例 6–2 フレンド宣言の問題の例 91
例 10–1 エラー状態のチェック 125
例 10–2 gcount の呼び出し 125
例 10–3 ユーザー定義の入出力操作 126
例 10–4 マルチスレッドでの安全性の無効化 126
例 10–5 マルチスレッドでの安全性の無効化 127
例 10–6 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 127
例 10–7 新しいクラス 128
例 10–8 新しいクラス階層 128
例 10–9 新しい関数 128
例 10–10 ロック処理の使用例 132
例 10–11 入出力操作とエラーチェックの不可分化 132
例 10–12 共有オブジェクトの破棄 134
例 10–13 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 134
例 13–1 string の抽出演算子 181
例 A–1 プリプロセッサのプログラム例 foo.cc 224
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 224
例 A–3 2 個のポインタを使用したループ 358

はじめに

『Oracle Solaris Studio 12.2 C++ ユーザーズガイド』では、Oracle Solaris Studio C++ コンパイラ、**cc** に関する環境とコマンド行オプションについて説明します。

このガイドは、C++ 言語と、Solaris または Linux オペレーティング環境に関する実用的な知識を持ち、Solaris Studio C++ コンパイラの効率的な使用法を学ぼうとしているプログラマを対象に書かれています。

サポートされるプラットフォーム

この Oracle Solaris Studio のリリースは、SPARC および x86 ファミリ (UltraSPARC、SPARC64、AMD64、Pentium、Xeon EM64T) プロセッサアーキテクチャーを使用するシステムをサポートしています。ご使用の Oracle Solaris オペレーティングシステムのバージョンに対するシステムのサポート状況は、ハードウェア互換性リスト (`http://www.sun.com/bigadmin/hcl`) を参照してください。ここには、すべてのプラットフォームごとの実装の違いについて説明されています。

このドキュメントでは、x86 関連の用語は次のものを指します。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品を指します。
- 「x64」は、AMD 64 または EM64T システムで、特定の 64 ビット情報を指します。
- 「32 ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

サポートされるシステムについては、ハードウェアの互換性に関するリストを参照してください。

Solaris Studio ドキュメントへのアクセス方法

ドキュメントには、次の場所からアクセスできます。

- ドキュメントは、次に示すドキュメント索引のページからアクセスできます。`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`。

- IDE、パフォーマンスアナライザ、dbxtool、および DLight の全コンポーネントのオンラインヘルプは、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログにある「ヘルプ」ボタンを使用してアクセスできます。

アクセシブルな製品ドキュメント

ドキュメントは、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ドキュメントを提供しております。アクセシブルなドキュメントは次の表に示す場所から参照することができます。

ドキュメントの種類	アクセシブルな形式と格納場所
マニュアル	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
『Oracle Solaris Studio 12.2 リリースの新機能』(以前はコンポーネントの README ファイル)	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
マニュアルページ	`man` コマンドを使用して Oracle Solaris 端末に表示
オンラインヘルプ	HTML 形式。IDE、dbxtool、DLight、パフォーマンスアナライザを使用時に「ヘルプ」メニュー、「ヘルプ」ボタン、および F1 キーから表示
リリースノート	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択

関連するサードパーティの **Web** サイトリファレンス

このマニュアルには、詳細な関連情報を提供するサードパーティの URL が記載されています。

注 – このマニュアルで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

開発者向けのリソース

http://www.oracle.com/technetwork/server-storage/solarisstudio を参照して、次の頻繁に更新されるリソースを確認してください。

- プログラミング技術とベストプラクティスに関する記事
- ソフトウェアのドキュメント、およびソフトウェアとともにインストールされる一連のドキュメント
- Oracle Solaris Studio ツールを使用して行う開発タスク全体を順を追って説明するチュートリアル
- サポートレベルに関する情報
- http://forums.sun.com/category.jspa?categoryID=113 にあるユーザーフォーラム

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1　表記上の規則

字体または記号	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、コード例を示します。	.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。 system%
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示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」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。

表 P–1 表記上の規則 (続き)

字体または記号	意味	例
\	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に、継続を示します。	`sun% grep '^#define \` `XV_VERSION_STRING'`

コード例は次のように表示されます。

- C シェル

  `machine_name% command y|n [filename]`

- C シェルのスーパーユーザー

  `machine_name# command y|n [filename]`

- Bourne シェルおよび Korn シェル

  `$ command y|n [filename]`

- Bourne シェルおよび Korn シェルのスーパーユーザー

  `# command y|n [filename]`

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

ドキュメント、サポート、およびトレーニング

追加リソースについては、次の Web サイトを参照してください。

- ドキュメント (http://docs.sun.com)
- サポート (http://www.oracle.com/us/support/systems/index.html)
- トレーニング (http://education.oracle.com) – 左のナビゲーションバーにある Sun リンクをクリックします。

ご意見の送付先

ドキュメントの品質や使いやすさに関するご意見やご提案をお待ちしています。間違いやその他の改善すべき箇所がありましたら、http://docs.sun.com で「Feedback」をクリックしてお知らせください。ドキュメント名とドキュメントの Part No.、および、可能な場合は章、節、ページ番号を記載してください。返答が必要な場合はお知らせください。

Oracle 技術ネットワーク (http://www.oracle.com/technetwork/index.html) では、Oracle ソフトウェアに関するさまざまなリソースを提供しています。

- 技術上の問題やソリューションについては、ディスカッションフォーラム (http://forums.oracle.com) を参照してください。
- 実践的なステップ・バイ・ステップのチュートリアルについては、Oracle By Example (http://www.oracle.com/technology/obe/start/index.html) を参照してください。
- サンプルコードのダウンロードについては、サンプルコード (http://www.oracle.com/technology/sample_code/index.html) を参照してください。

パート I

C++ コンパイラ

◆ ◆ ◆ 第 1 章

C++ コンパイラの紹介

この章では、最新の Solaris Studio C++ コンパイラの概要について説明します。

1.1 Solaris Studio 12.2 C++ 5.11 コンパイラの新機能

この節では、Solaris Studio 12.2 C++ 5.11 コンパイラリリースに導入された新機能と変更された機能の概要を一覧に示します。

- `-O` オプションまたは `-xO` オプションとともに `-g` オプションを使用すると、インライン化が有効になります。(246 ページの「A.2.30 -g」)
- C++ オプション `-xalias_level=compatible` オプションを使用すると、プログラムが C++ 標準の要件を満たすことが表明されます。(283 ページの「A.2.106 -xalias_level[=*n*]」)
- Oracle Solaris にインストールされた Apache C++ ライブラリのサポートが追加されました。(257 ページの「A.2.49 -library=*l*[,*l*...]」)
- `-compat=g` オプションにより、Gnu g++ との互換性が部分的に実現されます。(219 ページの「A.2.6 –compat[={ 4|5|g}]」)
- `-features=[no%]rvalueref` オプションにより、特定のコンパイラチェックが無効になります。(231 ページの「A.2.18 -features=*a*[,*a*...]」)
- SPARC-V9 ISA の SPARC VIS3 バージョンのサポートが追加されました。`-xarch=sparcvis3` オプションを使用してコンパイルすると、SPARC-V9 命令セットの命令、UltraSPARC および UltraSPARC-III 拡張機能、積和演算 (FMA) 命令、および VIS (Visual Instruction Set) バージョン 3.0 をコンパイラが使用できるようになります。(287 ページの「A.2.109 -xarch=*isa*」)
- x86 ベースのシステムに基づく `-xvector` オプションのデフォルト値が `-xvector=simd` に変更されました。(368 ページの「A.2.187 -xvector[=*a*]」)
- AMD SSE4a 命令セットのサポートが `-xarch=amdsse4a` オプションで使用できるようになりました。(287 ページの「A.2.109 -xarch=*isa*」)

- `-traceback` オプションを使用すると、重大なエラーが発生した場合に実行可能ファイルでスタックトレースを出力できます。(278 ページの「A.2.94 `-traceback[={ %none|common|signals_list}]`」)
- `-mt` オプションが `-mt=yes` または `-mt=no` に変更されています。(264 ページの「A.2.55 `-mt[={yes |no}]`」)
- `#warning` コンパイラ指令により、指令内のテキストが警告として発行され、コンパイルが続行されます。(42 ページの「2.5.4 警告とエラー」)
- 新しいプラグマ `does_not_read_global_data`、`does_not_write_global_data`、および `no_side_effect` が追加されました。(41 ページの「2.5.1 プラグマ」)
- ヘッダーファイル `mbarrier.h` を使用できるようになりました。このヘッダーファイルは、SPARC プロセッサと x86 プロセッサでマルチスレッドコードのさまざまなメモリーバリアー組み込み関数を定義します。(136 ページの「10.5 メモリーバリアー組み込み関数」)
- `-xprofile=tcov` オプションが拡張されて、オプションのプロファイルディレクトリパス名がサポートされるようになりました。また、`tcov` 互換のフィードバックデータも生成できます。(350 ページの「A.2.170 –xprofile=*p*」)
- このリリースでは、`-xMD` オプションと `-xMMD` オプション (C/C++) により記述された依存関係ファイルにより、既存のファイルがすべて上書きされます。(325 ページの「A.2.146 `-xMD`」)

1.2 x86 の特記事項

x86 Solaris プラットフォーム用にコンパイルを行う場合に注意が必要な、重要な事項がいくつかあります。

従来の Sun スタイルの並列化プラグマがすべてのプラットフォームで使用できなくなりました。代わりに OpenMP を使用してください。従来の並列化命令を OpenMP に変換する方法については、『Solaris Studio 12.2: OpenMP API ユーザーズガイド』を参照してください。

`-xarch` を `sse`、`sse2`、`sse2a`、または `sse3` 以降に設定してコンパイルしたプログラムは、必ずこれらの拡張子と機能を提供するプラットフォームでのみ実行してください。

Solaris 9 4/04 以降の Solaris OS リリースは、Pentium 4 互換プラットフォームでは SSE/SSE2 に対応しています。これより前のバージョンの Solaris OS は SSE/SSE2 に対応していません。`-xarch` で選択した命令セットが、実行中の Solaris OS で有効ではない場合、コンパイラはその命令セットのコードを生成またはリンクできません。

コンパイルとリンクを個別に行う場合は、必ずコンパイラを使ってリンクし、同じ `-xarch` 設定で正しい起動ルーチンがリンクされるようにしてください。

x86 の 80 ビット浮動小数点レジスタが原因で、x86 での演算結果が SPARC の結果と異なる場合があります。この差を最小にするには、`--fstore` オプションを使用するか、ハードウェアが SSE2 をサポートしている場合は `-xarch=sse2` でコンパイルします。

Solaris と Linux でも、固有の数学ライブラリ (たとえば、`sin(x)`) が同じではないため、演算結果が異なることがあります。

1.3 64 ビットプラットフォーム用のコンパイル

ILP32 32 ビットモデル用にコンパイルするには、`–m 32` オプションを使用します。ILP64 64 ビットモデル用にコンパイルするには、`–m64` オプションを使用します。

ILP 32 モデルでは、C++ 言語の int、long、およびポインタデータ型はすべて 32 ビット幅であることを指定します。long およびポインタデータ型を指定する LP64 モデルは、すべて 64 ビット拡張です。Solaris および Linux OS は、LP64 メモリーモデルの大きなファイルや配列もサポートします。

`-m64` を使用してコンパイルを行う場合、結果の実行可能ファイルは、64 ビットカーネルを実行する Solaris OS または Linux OS の 64 ビット UltraSPARC または x86 プロセッサでのみ動作します。コンパイル、リンク、および 64 ビットオブジェクトの実行は、64 ビット実行をサポートする Solaris または Linux OS でのみ行うことができます。

1.4 バイナリの互換性の妥当性検査

Solaris システムの Sun Studio 11 以降では、Solaris Studio コンパイラによってコンパイルされたプログラムのバイナリには、そのコンパイル済みバイナリによって想定されている命令セットを示すアーキテクチャーハードウェアフラグが付いています。実行時にこれらのマーカーフラグが検査され、実行しようとしているハードウェアで、そのバイナリが実行できることが確認されます。

プログラムにこれらのアーキテクチャーハードウェアフラグが含まれない場合、またはプラットフォームが適切な機能または命令セット拡張に対応していない場合、プログラムを実行することによりセグメント例外、または明示的な警告メッセージなしの不正な結果が発生することがあります。

このことは、`.il` インラインアセンブリ言語関数を使用しているプログラムや、SSE、SSE2、SSE2a、SSE3 の命令、およびより新しい命令と拡張機能を利用している `__asm()` アセンブラコードにも当てはまります。

1.5 準拠規格

この C++ コンパイラ (cc) は、ISO International Standard for C++, ISO IS 14882:2003, 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、この規格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 から出版された SPARC International による『SPARC アーキテクチャ・マニュアル バージョン 8』(トッパン刊) と『SPARC ArchitectureManual, Version 9』(ISBN 0-13-099227-5) (英語版のみ)に定義されています。

このマニュアルでは、「標準」は、前述の規格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」および「拡張」は、これらの規格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、それぞれの標準を策定する組織によって改訂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規格が改訂されたり、完全に書き換えられた場合、機能によっては、Solaris Studio 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。

1.6 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について説明しています。これは、『Oracle Solaris Studio 12.2 リリースの新機能』ガイドの一部となりました。次の内容が含まれています。

- 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

『新機能』ガイドには、このリリースのドキュメント索引 (http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation) からアクセスできます。

1.7 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。

```
example% man topic
```

C++ のドキュメント全体を通して、マニュアルページのリファレンスは、トピック名とマニュアルページの節番号で表示されます。CC(1) を表示するには、man CC と入力します。1 以外の節 (ieee_flags(3M) など) には、次のように man コマンドで -s オプションを使用してアクセスできます。

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

1.8 各国語のサポート

このリリースの 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、中国語、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、オペレーティングシステムのドキュメント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++ コンパイラを使用すると、次の作業を行うことができます。

- 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あと、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る。
- 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。

2.1 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細なリファレンスについては、付録 A 「C++ コンパイラオプション」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. エディタを使用して、表 2–1 に一覧表示されている有効な接尾辞の 1 つを指定し、C++ ソースファイルを作成する。
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.cc
```

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そのプログラム名にすることが一般的です。

あるいは、コンパイルのあとに 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、実行可能ファイルの名前を入力して、プログラムを実行します。

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

2.2 コンパイラの起動

本章のこれ以降の節では、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。

2.2.1 コマンド構文

コンパイラの一般的なコマンド行の構文を次に示します。

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

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

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

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

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
```

2.2.2 ファイル名に関する規則

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

表 2–1 C++ コンパイラが認識できるファイル名接尾辞

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

2.2.3 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。

2.3 バージョンが異なるコンパイラでのコンパイル

このコンパイラは、デフォルトではキャッシュを使用しません。キャッシュを使用するのは、-instances=extern が指定されているときだけです。キャッシュを使用する場合、コンパイラはキャッシュディレクトリのバージョンを調べ、その結果キャッシュバージョンに問題があることがわかると、エラーメッセージを出力します。将来の C++ コンパイラもキャッシュのバージョンを調べます。たとえば、将来のコンパイラは異なるテンプレートキャッシュのバージョン識別子を持っているため、現在のリリースで作成されたキャッシュディレクトリを処理しようとすると、次のようなエラーを出力します。

```
Template Database at ./SunWS_cache is incompatible with
this compiler
```

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。

コンパイラをアップグレードする際には、必ずキャッシュを消去するようにするとよいでしょう。テンプレートキャッシュディレクトリ (ほとんどの場合、テンプレートキャッシュディレクトリの名前は SunWS_cache) が入っているディレクトリすべてに対し、CCadmin -clean を実行します。CCadmin -clean の代わりに、rm -rf SunWS_cache と指定しても同様の結果が得られます。

2.4 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
```

2.4.1 コンパイルとリンクの流れ

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

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

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

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

2.4.2 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、次の手順でそれをほかのファイルとリンクします。

```
example% CC -c file1.cc          Make new object file
example% CC -o prgrm file1.o file2.o file3.o          Make executable file
```

リンク時には (2 行目)、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必ず 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)」エラーで、ルーチンがないために失敗します。

2.4.3 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、50 ページの「3.3.3 コンパイル時とリンク時のオプション」に示す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は非常に重大な意味を持ちます。

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 オプションまたは -m64 / -m32 オプションを使用してコンパイルする場合、これらの同じオプションをすべての cc コマンドに含める必要があります。
- -p、-xpg、-xprofile オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プロファイル処理ができなくなります。

- -g、-g0 オプションの場合、ある段階ではオプションを指定して別の段階では指定しないと、プログラムの正しさには影響はありませんが、プログラムを正しくデバッグできなくなります。これらのオプションでコンパイルされず、-g または -g0 でリンクされるいずれのモジュールも、デバッグには使用できません。--g オプション (または -g0 オプション)付きの main 関数があるモジュールをコンパイルするには、通常デバッグする必要があります。

次の例では、-library=stlport4 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。

```
 example% CC -library=stlport4 sbr.cc -c
example% CC -library=stlport4 main.cc -c
example% CC -library=stlport4 sbr.o main.o -o myprogram
```

-library=stlport4 を一貫して使用しない場合は、プログラムの特定の部分はデフォルトの libCstd を使用し、ほかの部分はオプションの置換である STLport ライブラリを使用します。結果として得られたプログラムは正常にリンクできず、どのような状況でも正常に動作しません。

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。

2.4.4 64 ビットメモリーモデル用のコンパイル

新しい -m64 オプションを使用して、対象コンパイルのメモリーモデルを指定します。結果の実行可能ファイルは、64 ビットカーネルを実行する Solaris OS または Linux OS の配下にある、64 ビットの UltraSPARC または x86 プロセッサでのみ動作します。コンパイルリンク、および 64 ビットオブジェクトの実行は、64 ビット実行をサポートする Solaris または Linux OS でのみ行うことができます。

2.4.5 コンパイラのコマンド行診断

-V オプションを指定すると、CC によって起動された各プログラムの名前とバージョン番号が表示されます。-v オプションを指定すると、CC によって起動されたコマンド行全体が表示されます。

–verbose=%all を指定すると、コンパイラに関する追加情報が表示されます。

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。

基本的には次のように区別されます。

- 認識できないオプション。これらの前にダッシュ (–) またはプラス記号 (+) が付けられ、警告が生成されます。
- 認識できない非オプション (先頭にダッシュかプラス符号 (+) が付いていないもの)には、警告が生成されません。ただし、リンカーへの引き渡しは行われます。リンカーが認識しない場合、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 is not a recognized CC option

CC: Warning: Option -bit passed to ld, if ld is invoked, ignored otherwise
```

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                    < - The user meant to type -fast
move.CC:
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors. No output written to a.out
```

2.4.6 コンパイラの構成

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

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

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

表 2–2 C++ コンパイルシステムの構成要素

コンポーネント	内容の説明	使用時の注意
ccfe	フロントエンド (コンパイラプリプロセッサ (前処理系) とコンパイラ)	
iropt	コードオプティマイザ	-xO[2-5]、-fast
ir2hf	x86: 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
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	x86: コードジェネレータ	-xO[2-5]、-fast
CClink	テンプレートのプリリンカー	-instances=extern オプションのみで使用します。
ld	リンクエディタ	

2.5 指示および名前の前処理

この節では、C++ コンパイラ特有の前処理の指示について説明します。

2.5.1 プラグマ

プリプロセッサ指令 pragma は C++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。C++ コンパイラが認識するプラグマ (指令) の詳細は、付録 B「プラグマ」を参照してください。

Solaris Studio C++ は、C99 のキーワードである _Pragma もサポートしています。次の 2 つの呼び出しは同等です。

```
#pragma dumpmacros(defs)
_Pragma("dumpmacros(defs)")
```

#pragma の代わりに _Pragma を使用するには、プラグマテキストをリテラル文字列として記述し、_Pragma キーワードの 1 つの引数として括弧で囲みます。

2.5.2 可変数の引数をとるマクロ

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

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

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

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

この結果は、次のようになります。

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

2.5.3 事前に定義されている名前

付録の 222 ページの「A.2.8 -D*name*[=*def*]」 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。

2.5.4 警告とエラー

#error および #warning プリプロセッサディレクティブを使用すると、コンパイル時の診断を生成できます。

#error *token-string*	エラー診断 *token-string* を発行して、コンパイルを終了します。
#warning *token-string*	警告診断 *token-string* を発行してコンパイルを続行します。

2.6 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 -x0*level* 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。最適化のレベルを下げてみてください。代わりに、最大のプロシージャを、個別のファイルに分割してください。

2.6.1 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
```

2.6.2 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ある必要があります。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
```

2.6.3 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 4G バイトに制限しています。

```
example$ ulimit -d 4000000
```

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 4G バイトに制限しています。

```
example% limit datasize 4G
```

どちらの例でも、オプティマイザは データ空間が 4G 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す。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ある必要があります。

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。

8G バイトのスワップ領域のあるマシンでは、次のコマンドを使用します。

sh シェルの場合

```
example$ ulimit -d 4000000
```

csh の場合

```
example% limit datasize 4G
```

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。

2.6.4 メモリー条件

ワークステーションには、少なくとも 2G バイトのメモリーを実装する必要があります。詳細な要件については、製品のリリースノートを参照してください。

2.7 C++ オブジェクトに対する strip コマンドの使用

Unix の strip コマンドは、C++ のオブジェクトファイルに対して使用すべきではありません。それらのオブジェクトファイルが使用不可能になることがあります。

2.8 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。

2.8.1 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
```

前述のコマンド 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
```

2.8.2 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 -m64'
```

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 変数が前述の例のように設定され、メイクファイル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、次と同じコンパイルが行われます。

```
CC -xO2 -m64 files...
```

2.8.3 make の使用

make ユーティリティーは、Solaris Studio 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。

2.8.3.1 make での CCFLAGS の使用

メイクファイル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。

◆ ◆ ◆ 第 3 章

3 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。オプションの詳細な説明は、216 ページの「A.2 オプションの一覧」を参照してください。

3.1 構文の概要

次の表は、一般的なオプション構文の形式の例です。

表 3–1 オプション構文形式の例

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

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。使用法の構文に関する詳細な説明は、「はじめに」の表記規則を参照してください。

3.2 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あと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 (ほか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)。この規則はリンカーのオプションには適用されません。
- `-features`、`-I-`、`-l`、`-L`、`-library`、`-pti`、`-R`、`-staticlib`、`-U`、`-verbose` および `-xprefetch` オプションで指定した内容は蓄積され、上書きはされません。
- `-D` オプションは累積されます。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。

3.3 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 「C++ コンパイラオプション」を参照してください。

これらのオプションは、特に記載がないかぎりすべてのプラットフォームに適用されます。Solaris SPARC システム版のオペレーティングシステムに特有の機能は *SPARC* として表記され、x86 システム版のオペレーティングシステムに特有の機能は *x86* として表記されます。

3.3.1 コード生成オプション

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します。
`+e{0|1}`	仮想テーブル生成を制御します。
`-g`	デバッグ用にコンパルします。
`-KPIC`	位置に依存しないコードを生成します。

表 3–2 コード生成オプション (続き)

オプション	処理
-Kpic	位置に依存しないコードを生成します。
-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。
-xaddr32	コードを 32 ビットアドレス空間に制限します (x86/x64)。
-xarch	ターゲットアーキテクチャーを指定します。
-xcode=*a*	(SPARC) コードのアドレス空間を指定します。
-Merge	(SPARC) データセグメントとテキストセグメントをマージします。
-xtarget	ターゲットシステムを指定します。
–xmodel	64 ビットオブジェクトの形式を Solaris x86 プラットフォーム用に変更します。
+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。
+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	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かぎりません。
-z *arg*	リンカーオプション

3.3.2 コンパイル時パフォーマンスオプション

表 3–3 コンパイル時パフォーマンスオプション

オプション	処理
-instlib	指定ライブラリにすでに存在しているテンプレートインスタンスの生成を禁止します。
-m32|-m64	コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。
-xinstrument	スレッドアナライザで分析するために、プログラムをコンパイルして計測します。

表 3–3 コンパイル時パフォーマンスオプション (続き)

オプション	処理
-xjobs	コンパイラが処理を行うために作成するプロセスの数を設定します。
-xpch	共通の一連のインクルードファイル群を共有するソースファイルを持つアプリケーションのコンパイル時間を短縮できます。
-xpchstop	-xpch でプリコンパイル済みヘッダーファイルを作成する際に適用される、最後のインクルードファイルを指定します。
-xprofile_ircache	(SPARC) -xprofile=collect で保存されたコンパイルデータを再使用します。
-xprofile_pathmap	(SPARC) 1 つのプロファイルディレクトリに存在する複数のプログラムや共有ライブラリをサポートします。

3.3.3 コンパイル時とリンク時のオプション

次の表は、リンク時とコンパイル時の両方に指定する必要があるオプションを一覧表示します。

表 3–4 コンパイル時とリンク時のオプション

オプション	処理
-fast	実行可能コードの速度を向上させるコンパイルオプションの組み合わせを選択します。
-m32|-m64	コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。
-mt	--D_REENTRANT --lthread に展開されるマクロオプションです。
-xarch	命令セットアーキテクチャーを指定します。
-xautopar	複数プロセッサ用の自動並列化を有効にします。
-xhwcprof	(SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。
-xipo	内部手続き解析パスを呼び出すことにより、プログラム全体の最適化を実行します。
-xlinkopt	再配置可能なオブジェクトファイルのリンク時の最適化を実行します。

表 3–4 コンパイル時とリンク時のオプション (続き)

オプション	処理
-xmemalign	(SPARC) メモリーの予想される最大境界整列と境界整列していないデータアクセスの動作を指定します。
-xopenmp	明示的な並列化のための OpenMP インタフェースをサポートします。このインタフェースには、ソースコード指令セット、実行時ライブラリルーチン、環境変数などが含まれます。
-xpagesize	スタックとヒープの優先ページサイズを設定します。
-xpagesize_heap	ヒープの優先ページサイズを設定します。
-xpagesize_stack	スタックの優先ページサイズを設定します。
-xpg	gprof(1) 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します。
-xprofile	プロファイルのデータを収集、または最適化のためにプロファイルを使用します。
-xvector=lib	ベクトルライブラリ関数を自動呼び出しします。

3.3.4 デバッグオプション

表 3–5 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ません。
-dryrun	ドライバがコンパイラに渡すオプションを表示しますが、コンパイルはしません。
-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 に出力しますが、コンパイルはしません。
-g	デバッグ用にコンパルします。
-g0	デバッグ用にコンパイルしますが、インライン機能は無効にしません。
-H	インクルードされるファイルのパス名を出力します。
-keeptmp	コンパイル中に作成されたすべての一時ファイルを残します。

表 3–5 デバッグオプション (続き)

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します。
`-P`	ソースの前処理だけを行い、`.i` ファイルに出力します。
`-Qoption`	オプションをコンパイル中の各処理に直接渡します。
`-readme`	`README` ファイルの内容を表示します。
`-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。
`-temp=`*dir*	一時ファイルのディレクトリを指定します。
`-verbose=`*vlst*	コンパイラの冗長性を制御します。
`-xcheck`	スタックオーバーフローの実行時検査を追加します。
`-xdumpmacros`	定義内容、定義および解除された位置、使用されている場所に関する情報を出力します。
`-xe`	構文と意味のエラーのチェックだけを行います。
`-xhelp=flags`	コンパイラオプションの要約を一覧表示します。
`-xport64`	32 ビットアーキテクチャーから 64 ビットアーキテクチャーへの移植中の一般障害について警告します。

3.3.5 浮動小数点オプション

表 3–6 浮動小数点オプション

オプション	処理
`-fma`	(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。
`-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。
`-fprecision=`*p*	*x86*: 浮動小数点精度モードを設定します。
`-fround=`*r*	起動時に IEEE 丸めモードを有効にします。
`-fsimple=`*n*	浮動小数点最適化の設定を行います。

表 3–6 浮動小数点オプション (続き)

オプション	処理
`-fstore`	*x86*: 浮動小数点式の精度を強制的に使用します。
`-ftrap=`*tlst*	起動時に IEEE トラップモードを有効にします。
`-nofstore`	*x86*: 強制された式の精度を無効にします。
`-xlibmieee`	例外時に `libm` が数学ルーチンに対し IEEE 754 値を返します。

3.3.6 言語オプション

表 3–7 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します。
`-features=`*alst*	C++ の各機能を有効化または無効化します。
`-xchar`	文字型が符号なしと定義されているシステムからのコードの移行を容易に行えるようにします。
`-xldscope`	共有ライブラリをより速くより安全に作成するため、変数と関数の定義のデフォルトリンカースコープを制御します。
`-xthreadvar`	(SPARC) デフォルトのスレッドローカルな記憶装置へのアクセスモードを変更します。
`-xtrigraphs`	文字表記シーケンスを認識します。
`-xustr`	16 ビット文字で構成された文字リテラルを認識します。

3.3.7 ライブラリオプション

表 3–8 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します。
`-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。
`-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。

表 3–8 ライブラリオプション (続き)

オプション	処理
-h*name*	生成される動的共有ライブラリに内部名を割り当てます。
-i	ld(1) がどのような LD_LIBRARY_PATH 設定も無視します。
-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します。
-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 を追加します。
-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みます。
-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。
-norunpath	ライブラリのパスを実行可能ファイルに組み込みません。
-R*plst*	動的ライブラリの検索パスを実行可能ファイルに組み込みます。
-staticlib=*llst*	静的にリンクする C++ ライブラリを指定します。
-xar	アーカイブライブラリを作成します。
-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します。
-xia	(Solaris) 適切な区間演算ライブラリをリンクし、浮動小数点環境を設定します。
-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します。
-xlibmieee	例外時に libm が数学ルーチンに対し IEEE 754 値を返します。
-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します。
-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。
-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。
-xnolibmil	コマンド行の -xlibmil を取り消します。
-xnolibmopt	数学ルーチンのライブラリを使用しません。

3.3.8 廃止オプション

注 – 次のオプションは、現在は廃止されているためにコンパイラに受け入れられないか、将来のリリースでは削除されます。

表 3–9 廃止オプション

オプション	処理
-library=%all	将来のリリースで削除されます。
-xlic_lib=sunperf	Sun Performance Library にリンクするには、–library=sunperf を使用します。
-xlicinfo	非推奨。
-noqueue	ライセンス情報のキューイングを行いません。
-ptr	コンパイラは無視します。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ります。
-sb、–sbfast、–xsb、–xsbfast	廃止され、メッセージを表示されずに無視されます。
-vdelx	将来のリリースで削除されます。
-x386	適切な –xtarget オプションを使用します。
-x486	適切な –xtarget オプションを使用します。
-xcg89	-xtarget=ss2 を使用します。
-xcrossfile	代わりに -xipo を使用してください。
-xnativeconnect	廃止。これに代わるオプションはありません。
-xprefetch=yes	代わりに - xprefetch=auto,explicit を使用します。
-xprefetch=no	代わりに -xprefetch=no%auto,no%explicit を使用します。
-xvector=yes	代わりに、--xvector=lib を使用します。
-xvector=no	代わりに、-xvector=none を使用します。

3.3.9 出力オプション

表 3–10 出力オプション

オプション	処理
`-c`	コンパイルのみ。オブジェクト (`.o`) ファイルを作成しますが、リンクはしません。
`-dryrun`	ドライバからコンパイラに対して発行されたコマンド行を表示しますが、コンパイルを行いません。
`-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` に出力しますが、コンパイルはしません。
`-erroff`	コンパイラの警告メッセージを抑止します。
`-errtags`	各警告メッセージのメッセージタグを表示します。
`-errwarn`	指定の警告メッセージが出力されると、`cc` はエラーステータスを返して終了します。
`-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します。
`-H`	インクルードされるファイルのパス名を出力します。
`-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します。
`-o` *filename*	出力ファイルや実行可能ファイルの名前を *filename* にします。
`-P`	ソースの前処理だけを行い、`.i` ファイルに出力します。
`-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。
`-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。
`-verbose=`*vlst*	コンパイラの冗長性を制御します。
`+w`	必要に応じて追加の警告を出力します。
`+w2`	該当する場合は、より多くの警告を出力します。
`-w`	警告メッセージを抑止します。
`-xdumpmacros`	定義内容、定義および解除された位置、使用されている場所に関する情報を出力します。

表 3–10 出力オプション (続き)

オプション	処理
-xe	ソースファイルの構文と意味のチェックだけを行い、オブジェクトや実行可能コードの出力はしません。
-xhelp=flags	コンパイラオプションの要約を一覧表示します。
-xhelp=readme	README ファイルの内容を表示します。
-xM	メイクファイルの依存情報を出力します。
-xM1	依存情報の生成は行いますが、/usr/include の組み込みはしません。
-xtime	コンパイル処理ごとの実行時間を報告します。
-xwe	すべての警告をエラーに変換します。
-z *arg*	リンカーオプション

3.3.10 実行時パフォーマンスオプション

表 3–11 実行時パフォーマンスオプション

オプション	処理
-fast	一部のプログラムで最適な実行速度が得られるコンパイルオプションの組み合わせを選択します。
-fma	(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。
-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します。
-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。
-m32|-m64	コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。
-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ー命令セットを指定します。
-xbinopt	あとで最適化、変換、分析を行うために、バイナリを準備します。
-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します。

表 3–11 実行時パフォーマンスオプション (続き)

オプション	処理
`-xcache=`*c*	(SPARC) オプティマイザのターゲットキャッシュプロパティーを定義します。
`-xcg89`	汎用の SPARC V7 アーキテクチャー用のコンパイルを行います。
`-xcg92`	SPARC V8 アーキテクチャー用のコンパイルを行います。
`-xchip=`*c*	ターゲットのプロセッサチップを指定します。
`-xF`	リンカーによる関数と変数の順序変更を有効にします。
`-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します。
`-xipo`	内部手続きの最適化を実行します。
`-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します。
`-xlibmopt`	最適化された数学ルーチンライブラリを使用します。
`-xlinkopt`	(SPARC) オブジェクトファイル内のあらゆる最適化のほかに、結果として出力される実行可能ファイルや動的ライブラリのリンク時最適化も行います。
`-xmemalign=`*ab*	(SPARC) メモリーの予想される最大境界整列と境界整列していないデータアクセスの動作を指定します。
`-xnolibmil`	コマンド行の `-xlibmil` を取り消します。
`-xnolibmopt`	数学ルーチンのライブラリを使用しません。
`-xO`*level*	最適化レベルを *level* にします。
`-xpagesize`	スタックとヒープの優先ページサイズを設定します。
`-xpagesize_heap`	ヒープの優先ページサイズを設定します。
`-xpagesize_stack`	スタックの優先ページサイズを設定します。
`-xprefetch`[`=`*lst*]	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します。
`-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します。
`-xprofile`	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します。
`-xregs=`*rlst*	一時レジスタの使用を制御します。

表 3–11 実行時パフォーマンスオプション (続き)

オプション	処理
`-xsafe=mem`	(SPARC) メモリーに関するトラップを起こさないものとします。
`-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*	ターゲットの命令セットと最適化のシステムを指定します。
`-xthreadvar`	デフォルトのスレッドローカル記憶装置アクセスモードを変更します。
`-xunroll=`*n*	可能な場合は、ループを展開します。
`-xvis`	(SPARC) VIS 命令セットに定義されているアセンブリ言語テンプレートをコンパイラが認識します。

3.3.11 プリプロセッサオプション

表 3–12 プリプロセッサオプション

オプション	処理
`-D`*name*[`=`*def*]	シンボル *name* をプリプロセッサに定義します。
`-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` に出力しますが、コンパイルはしません。
`-H`	インクルードされるファイルのパス名を出力します。
`-P`	ソースの前処理だけを行い、`.i` ファイルに出力します。
`-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。
`-xM`	メイクファイルの依存情報を出力します。
`-xM1`	依存情報を生成しますが、`/usr/include` は除きます。

3.3.12 プロファイルオプション

表 3–13 プロファイルオプション

オプション	処理
`-p`	`prof` でプロファイル処理するためのデータを収集するオブジェクトコードを用意します。

表 3–13 プロファイルオプション (続き)

オプション	処理
-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します。
-xprofile	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します。

3.3.13 リファレンスオプション

表 3–14 リファレンスオプション

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

3.3.14 ソースオプション

表 3–15 ソースオプション

オプション	処理
-H	インクルードされるファイルのパス名を出力します。
-I*pathname*	include ファイル検索パスに *pathname* を追加します。
-I-	インクルードファイル検索規則を変更します。
-xM	メイクファイルの依存情報を出力します。
-xM1	依存情報を生成しますが、/usr/include は除きます。

3.3.15 テンプレートオプション

表 3–16 テンプレートオプション

オプション	処理
-instances=*a*	テンプレートインスタンスの位置とリンケージを制御します。

表 3–16 テンプレートオプション　(続き)

オプション	処理
`-template=`*wlst*	さまざまなテンプレートオプションを有効または無効にします。

3.3.16 スレッドオプション

表 3–17 スレッドオプション

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

パート II

C++プログラムの作成

◆ ◆ ◆ 第 4 章

4 言語拡張

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

`-features=extensions` オプションを使用すると、ほかの 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。

4.1 リンカースコープ

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

これらの指示子を使うと、リンカースコープのマップファイルは使用しなくてすみます。 `-xldscope` をコマンド行で指定することによって、変数スコープのデフォルト設定を制御することもできます。

詳細は、319 ページの「A.2.136 -xldscope={v}」 を参照してください。

表 4–1 リンカースコープ宣言指定子

値	意味
__global	シンボル定義には大域リンカースコープとなります。これは、もっとも限定的でないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。
__symbolic	シンボル定義は、シンボリックリンカースコープとなります。これは、大域リンカースコープより限定的なリンカースコープです。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュールの外側では、シンボルは大域と同じです。このリンカースコープはリンカーオプション -Bsymbolic に対応します。C++ ライブラリでは -Bsymbolic を使用できませんが、__symbolic 指定子は問題なく使用できます。リンカーの詳細については、ld(1) を参照してください。
__hidden	シンボル定義は、隠蔽リンカースコープとなります。隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。

より限定的な指定子を使ってシンボル定義を宣言しなおすことはできますが、より限定的でない指定子を使って宣言しなおすことはできません。シンボルは一度定義したら、異なる指示子で宣言することはできません。

__global はもっとも制限の少ないスコープです。__symbolic はより制限されたスコープです。__hidden はもっとも制限の多いスコープです。

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

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

4.1.1 Microsoft Windows との互換性

動的ライブラリに関して Microsoft Visual C++ (MSVC++) に含まれる類似のスコープ機能との互換性を保つため、次の構文もサポートされています。

`__declspec(dllexport)` は `__symbolic` と同一です。

`__declspec(dllimport)` は `__global` と同一です。

Solaris Studio C++ でこの構文の利点を活用するには、`-xldscope=hidden` オプションを `CC` コマンド行に追加する必要があります。結果は、MSVC++ を使用する場合と比較可能なものになります。MSVC++ を使用する場合は、定義ではなく外部シンボルの宣言のみに関して `__declspec(dllimport)` が使用されることが想定されます。次に例を示します。

```
__declspec(dllimport) int foo(); // OK
__declspec(dllimport) int bar() { ... } // not OK
```

MSVC++ は、定義に対する `dllimport` の許容が緩やかであり、Solaris Studio C++ を使用する場合の結果と異なります。特に、Solaris Studio C++ で定義に対して `dllimport` を使用する場合の結果は、シンボルがシンボリックリンケージではなくグローバルリンケージを持つことになります。Microsoft Windows 上の動的ライブラリは、シンボルのグローバルリンケージをサポートしません。この問題が発生している場合は、定義に対して dllimport ではなく dllexport を使用するようにソースコードを変更できます。その後、MSVC++ と Solaris Studio C++ で同じ結果を得ることができます。

4.2 スレッドローカルな記憶装置

スレッドローカルの変数を宣言して、スレッドローカルな記憶領域を利用します。スレッドローカルな変数の宣言は、通常の変数宣言に宣言指定子 `__thread` を加えたものです。詳細については、364 ページの「A.2.182 -xthreadvar[=*o*]」 を参照してください。

`__thread` 指定子は、スレッド変数の最初の宣言部分に含める必要があります。`__thread` 指定子で宣言した変数は、`__thread` 指定子がない場合と同じように結合されます。

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

とはできますが、 `__thread int x = f();` を使用することはできません。スレッド変数には、特殊なコンストラクタやデストラクタを持たせるべきではありません。とくに `std::string` 型をスレッド変数として持たせることはできません。

スレッド変数の演算子 (&) のアドレスは、実行時に評価され、現在のスレッドの変数のアドレスが返されます。したがって、スレッド変数のアドレスは定数ではありません。

スレッド変数のアドレスは、対応するスレッドの有効期間の間は安定しています。変数の有効期間内は、プロセス内の任意のスレッドがスレッド変数のアドレスを自由に使用できます。スレッドが終了したあとは、スレッド変数のアドレスを使用できません。スレッドの変数のアドレスは、スレッドが終了するとすべて無効となります。

4.3 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る必要があります。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、前述の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。

4.4 enum の型と変数の前方宣言の実行

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

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

```
enum E; // invalid: forward declaration of enum not allowed
E e;    // invalid: type E is incomplete
```

enum 定義では、ほかの enum 定義を参照できず、ほか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ーでは、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。

4.5 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、-features=extensions 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; // invalid, E is incomplete
```

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。

4.6 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; // invalid: E is not a scope name
```

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。enum 型の名前だった場合に、-features=extensions オプションはコンパイラにスコープ修飾子を無視するよう命令します。

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。

4.7 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。

次は、-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;
  };  // invalid: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // invalid: anonymous struct
};
```

これらの構造体のメンバー名は、構造体名で修飾しなくても認識されます。たとえば、共用体 U が前述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;
```

4.8 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)); // invalid
}
```

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。

4.9 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おり、また、すべての定義が同一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る必要があります。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。

4.10 事前定義済み __func__ シンボルの関数名としての使用

コンパイラでは、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
```

識別子 __FUNCTION__ も定義され、__func__ と同等になります。

4.11 サポートされる属性

__attribute__ ((*keyword*)) によって、または [[*keyword*]] によって代わりに呼び出される次の属性は、互換性のためにコンパイラによって実装されます。

always_inline — 次と同等: #pragma inline および -xinline
noinline — 次と同等: #pragma no_inline および -xinline
pure — 次と同等: #pragma does_not_write_global_data
const — 次と同等: #pragma no_side_effect
malloc — 次と同等: #pragma returns_new_memory
constructor — 次と同等: #pragma init

destructor — 次と同等: #pragma fini
weak — 次と同等: #pragma weak
noreturn — 次と同等: #pragma does_not_return
visibility
returns_twice
packed — 下記を参照
atomic
outer
relaxed
mode
aligned
strong

4.11.1 __packed__

struct または union の型定義に添付されるこの属性は、必要なメモリーを最小限に抑えるために、構造体または共用体の各メンバー (幅が 0 のビットフィールドを除く) の配置を指定します。enum 定義に添付する場合は、この属性は最小の整数型を使用することを示します。

struct および union 型に対してこの属性を指定する場合は、構造体または共用体の各メンバーに対して packed 属性を指定する場合と同じことを意味します。

次の例では、struct my_packed_struct のメンバーは互いに隣接してパックされますが、メンバーの内部レイアウトはパックされません。内部レイアウトをパックするには、struct my_unpacked_struct もパックする必要があります。

```
struct my_unpacked_struct
{
   char c;
   int i;
;

struct __attribute__ ((__packed__)) my_packed_struct
{
   char c;
   int  i;
   struct my_unpacked_struct s;
};
```

この属性を指定できるのは、enum、struct、または union の定義のみです。列挙型、構造体、共用体のいずれも定義しない typedef に対して、この属性は指定できません。

◆ ◆ ◆ 第 5 章

5 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。

5.1 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。

5.1.1 言語に対応したヘッダーファイル

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

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

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

```
#else
int function();               // K&R C
#endif
```

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

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

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

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

ARM C++ では、__cplusplus マクロの値は 1 です。ISO C++ では、このマクロの値は 199711L (long 定数で表現した、規格の年と月) です。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。

```
// template function specialization
#if __cplusplus < 199711L
int power(int,int);                       // ARM C++
#else
template <> int power(int,int);           // ISO C++
#endif
```

5.1.2 べき等ヘッダーファイル

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

```
#ifndef HEADER_H
#define HEADER_H
```

```
/* contents of header file */
#endif
```

5.2 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。

5.2.1 テンプレート定義の取り込み

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場合、編成が定義の取り込みです。次に例を示します。

main.cc

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

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

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 – テンプレートヘッダーをべき等にすることは非常に重要です。76 ページの「5.1.2 べき等ヘッダーファイル」を参照してください。

5.2.2 テンプレート定義の分離

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

twice.h

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

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。76 ページの「5.1.2 べき等ヘッダーファイル」を参照してください。なお、テンプレートの定義分離型編成は、すべてのコンパイラでサポートされているわけではありません。

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

注 – 通常、テンプレート定義ファイルには、ソースファイルの拡張子 (.c、.C、.cc、.cpp、.cxx、.c++ のいずれか) を付けますが、このテンプレート定義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、105 ページの「7.5 テンプレート定義の検索」を参照してください。

-E オプションまたは -P オプションを使用してプリプロセッサ出力を生成する場合、定義分離ファイルの構成では、テンプレート定義を .i ファイルに含めることが許可されません。見つからない定義があるため、.i ファイルのコンパイルに失敗します。テンプレート定義ファイルをテンプレート宣言ヘッダー (次のコード例を参照) に条件付きで含めることで、コマンド行で -template=no%extdef を使用することによりテンプレート定義を使用できます。libCtd ライブラリと STLport ライブラリは、この方法で実装されます。

```
// templace declaration file
template <class T> class foo { ... };
#ifdef _TEMPLATE_NO_EXTDEF
#include "foo.cc"  //template definition file
#endif
```

ただし、マクロ _TEMPLATE_NO_EXTDEF を自分で定義しないでください。–template=no%extdef オプションなしで定義すると、テンプレート定義ファイルが複数含められるためにコンパイルエラーが発生することがあります。

◆ ◆ ◆ 第 6 章

6 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。

6.1 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。

6.1.1 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する必要があります。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が提供されますが、テンプレートを実装するにはほかの情報も必要です。

```
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。

6.1.2 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ある必要があります。この制限は「単一定義ルール」と呼ばれています。

6.1.3 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ほかのすべての関数と同様に使用することができます。テンプレートの使用は、そのテンプレートの命名と関数引数の提供で構成されます。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(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

6.2 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またはデータ型を記述します。クラステンプレートに記述されているクラスは、型、整数値、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。

6.2.1 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;
```

6.2.2 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する必要があります。

```
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

6.2.3 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。

6.2.3.1 関数メンバーの定義

テンプレート関数メンバーの定義は、テンプレートパラメータの指定と、それに続く関数定義から構成されます。関数識別子は、クラステンプレートのクラス名とそ

のテンプレートの引数で修飾されます。次の例は、template <class Elem> というテンプレートパラメータ指定を持つ Array クラステンプレートの 2 つの関数メンバー定義を示しています。それぞれの関数識別子は、テンプレートクラス名とテンプレート引数 Array<Elem> で修飾されています。

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

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

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

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

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

6.2.3.2 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;
```

6.2.4 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 (85 ページの「6.3 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、ほか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( );
.
```

6.3 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>` クラスと `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82 ページの「6.2 クラステンプレート」の節に示す `Array<int>` の例では、`Elem` が表示されるたびに、コンパイラが `int` に置き換えられます。

6.3.1 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。

6.3.2 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。

6.3.2.1 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あと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);
```

6.3.2.2 テンプレートクラスの明示的インスタンス化

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

6.3.2.3 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あと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();
```

6.3.2.4 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あとにテンプレート引数を指定します。

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

6.4 テンプレートの編成

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

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

また、次のように定義することができます。

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

前述の例は、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);   // construct an array of ints
sort(int_array);             // sort it
```

6.5 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ある必要があります。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。

6.6 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。

6.6.1 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する必要があります。次の例は *twice* と *sort* の特殊化された実装を宣言しています。

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

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

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

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

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

6.6.2 テンプレートの特殊化定義

宣言するテンプレートの特殊化はすべて定義する必要があります。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;}}
```

6.6.3 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ほか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。

6.6.4 部分特殊化

前述の例では、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 {...}; //primary template
template<> class A<int, double> {...};    //specialization
```

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<class U> class A<int> {...};          // Example 1
template<class T, class U> class A<T*> {...}; // Example 2
template<class T> class A<T**, char> {...};   // Example 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。

6.7 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。

6.7.1 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
// Common template definition
template <class Source, class Target>
Target converter(Source source)
       {temporary = (intermediary)source;
       return (Target)temporary;}
use1.cc
typedef int intermediary;
int temporary;

#include "use_common.h"
use2.cc
typedef double intermediary;
unsigned int temporary;

#include "use_common.h"
```

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

6.7.2 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。

例 6–1 テンプレート引数としての局所型の問題の例

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

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

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

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

`file1.cc` に登録された `Foo` 型は、`file2.cc` 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。

6.7.3 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る必要があります。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る必要があります。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。

例 6–2 フレンド宣言の問題の例

```
array.h
// generates undefined error for the operator<< function
#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>
```

```
// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

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

6.7.4 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; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;                 // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

6.7.5 テンプレート名の入れ子

「>>」 という文字連続型は右シフト演算子と解釈されるため、あるテンプレート名を別のテンプレート名で使用する場合は注意が必要です。隣接する「>」文字を少なくとも 1 つの空白文字で区切ってください。

次に誤った書式の例を示します。

```
Array<String<10>> short_string_array(100); // >> = right-shift
```

前述の文は、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);
```

6.7.6 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する場合は、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する場合は、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する場合は、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。

6.7.7 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

`-instances=extern` を指定して複数のプログラムまたはライブラリを構築する場合は、それらを別のディレクトリに構築することを推奨します。同一ディレクトリ内に構築する場合は、構築ごとにリポジトリを消去する必要があります。これにより、予期しないエラーが回避されます。詳細は、104 ページの「7.4.4 テンプレートリポジトリの共有」を参照してください。

次のような各ファイルが存在する、次の例を考慮してください。`make` ファイル、`a.cc`、`b.cc`、`x.h`、および `x.cc` この例は、`-instances=extern` を指定する場合にのみ意味があることに注意してください。

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

```
all: a b

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

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

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

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

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

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

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

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

int main()
{
  X<int> temp1;

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

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

int 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。次のコマンドでは、エラーが発生します。

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

次のコマンドでは、エラーは発生しません。

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

◆ ◆ ◆ 第 7 章

7 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ます。コンパイラは、テンプレートリポジトリを使って、別々のコンパイル間でテンプレートインスタンスを共有することができます。また、テンプレートコンパイルのいくつかのオプションを使用できます。コンパイラは、別々のソースファイルにあるテンプレート定義を見つけ、テンプレートインスタンスと main コード行の整合性を維持する必要があります。

7.1 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、コンパイラは通知しません。そのほかに、`+w` オプション を指定すると、テンプレートのインスタンス化が行われたときに問題になりそうな内容が通知される場合があります。

7.2 リポジトリの管理

`CCadmin`(1) コマンドは、テンプレートリポジトリを管理します (`-instances=extern` オプションを使用する場合のみ)。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` の `-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。

7.2.1 生成されるインスタンス

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

7.2.2 全クラスインスタンス化

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

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

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

ISO C++ 標準では、特定のテンプレート引用により、すべてのメンバーが正当であるとはかぎらないテンプレートクラスを作成してよいと規定しています。不正メンバーをインスタンス化しないかぎり、プログラムは依然として適正です。ISO C++ 標準ライブラリでは、この技法が使用されています。ただし、`-template=wholeclass` オプションはすべてのメンバーをインスタンス化するので、問題のあるテンプレート引数を使ってインスタンス化する場合には、この種のテンプレートクラスに使用できません。

7.2.3 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。

7.2.4 テンプレートインスタンスの配置とリンケージ

デフォルトでは、インスタンスは特別なアドレスセクションに移動し、リンカーは重複を認識、および破棄します。コンパイラには、インスタンスの配置とリンケージの方法として、外部、静的、大域、明示的、半明示的のどれを使うかを指定できます。

- 外部 インスタンスは、次の条件が成立する場合に最大のパフォーマンスを達成します。
  - プログラムに含まれているインスタンス全体は小さいが、各コンパイル単位がそれぞれ参照するインスタンスが大きい。
  - 2、3 個以上のコンパイル単位で参照されるインスタンスがほとんどない。

  静的、非推奨。次を参照してください。
- デフォルトである大域インスタンスは、あらゆる開発に適していますが、さまざまなインスタンスをオブジェクトが参照する場合に最適です。
- 明示的インスタンスは、厳密に管理されたアプリケーションコンパイル環境に適しています。
- 半明示的インスタンスは、前述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、用途はかぎられています。

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、85 ページの「6.3 テンプレートのインスタンス化」にあります。

7.3 外部インスタンス

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

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

注 – コンパイルとリンクを別々に実行し、コンパイル処理で -instance=extern を指定する場合は、リンク処理でも -instance=extern を指定する必要があります。

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

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

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

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

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

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

詳細は、表 14–3を参照してください。

7.3.1 キャッシュの衝突

-instance=extern を指定する場合、キャッシュの衝突の可能性があるため、異なるバージョンのコンパイラを同一ディレクトリ内で実行しないでください。-instances=extern テンプレートモデルを使用する場合は、次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。すべてのバイナリ (.o、a、.so、実行可能プログラム) は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。ほか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。メイクファイル内での依存により、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。

7.3.2 静的インスタンス

注 – -instances=static オプションは、非推奨です。-instances=global が static の利点をすべて備えており、かつ欠点を備えていないので、-instances=static を使用する理由はなくなっています。このオプションは、今はもう存在していない問題を克服するために、以前のバージョンで提供されました。

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

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

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

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

注 – プログラムがコンパイル単位間で、テンプレートクラスまたはテンプレート機能の静的データメンバーなどの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。

7.3.3 大域インスタンス

旧リリースのコンパイラとは異なり、新リリースでは、大域インスタンスの複数のコピーを防ぐ必要はありません。

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

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

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

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

7.3.4 明示的インスタンス

明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されたテンプレートに対してのみ生成されます。暗黙的なインスタンス化は行われません。インスタンスは現在のコンパイル単位に置かれます。

この方法の利点はテンプレートのコンパイル量もオブジェクトのサイズも、ほかのどの方法より小さくて済むことです。

欠点は、すべてのインスタンス化を手動で行う必要がある点です。

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

明示的リンケージは、`-instances=explicit` オプションによって指定します。

7.3.5 半明示的インスタンス

半明示的インスタンスの場合、インスタンスは、明示的にインスタンス化されるテンプレートやテンプレート本体の中で暗黙的にインスタンス化されるテンプレートに対してのみ生成されます。明示的に作成されるインスタンスが必要とするインス

タンスは自動的に生成されます。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ます。インスタンスが大域リンケージを受けることはなく、テンプレートリポジトリには保存されません。

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

7.4 テンプレートリポジトリ

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

7.4.1 リポジトリの構造

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

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

7.4.2 テンプレートリポジトリへの書き込み

コンパイラは、テンプレートインスタンスを格納しなければならないとき、出力ファイルに対応するテンプレートリポジトリにそれらを保存します。たとえば、次のコマンド行では、オブジェクトファイルを `./sub/a.o` に、テンプレートインスタンスを `./sub/SunWS_cache` 内のリポジトリにそれぞれ書き込みます。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
```

7.4.3 複数のテンプレートリポジトリからの読み取り

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。つまり、次のコマンド行は、/sub1/SunWS_cache と /sub2/SunWS_cache を読み取り、必要な場合は ./SunWS_cache に書き込みます。

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

7.4.4 テンプレートリポジトリの共有

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

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

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

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

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

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

注 – テンプレートのソースコードを削除する場合や、テンプレートの使用を停止する場合も、テンプレートのインスタンスはキャッシュ内にとどまります。関数テンプレートの署名を変更する場合も、古い署名を使用しているインスタンスはキャッシュ内にとどまります。これらの課題が原因でコンパイル時またはリンク時に予期しない動作が発生した場合は、テンプレートキャッシュをクリアし、プログラムを再構築してください。

7.5 テンプレート定義の検索

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

定義の検索はかなり複雑で、エラーを発生しやすい傾向があります。このため、可能であれば、定義取り込み型のテンプレートファイルの編成を使用したほうがよいでしょう。こうすれば、定義検索をまったく行わなくて済みます。77 ページの「5.2.1 テンプレート定義の取り込み」を参照してください。

注 – `--template=no%extdef` オプションを使用する場合、コンパイラは分離されたソースファイルを検索しません。

7.5.1 ソースファイルの位置規約

オプションファイルで提供されるような特定の指令がない場合には、コンパイラは `Cfront` 形式の方法でテンプレート定義ファイルを検出します。この方法の場合、テンプレート宣言ファイルと同じベース名がテンプレート定義ファイルに含まれている必要があります。また、テンプレート定義ファイルが現在の `include` パス上に存在している必要もあります。たとえば、テンプレート関数 `foo()` が `foo.h` 内にある場合には、それと一致するテンプレート定義ファイルの名前を `foo.cc` か、またはほ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。

7.5.2 定義検索パス

`-I` で設定する通常の検索パスの代わりに、`-pti`*directory* オプションでテンプレート定義ファイルの検索ディレクトリを指定することができます。複数の `-pti` フラグは、複数の検索ディレクトリ、つまり 1 つの検索パスを定義します。`-pti`*directory* を

使用している場合には、コンパイラはこのパス上のテンプレート定義ファイルを探し、-I フラグを無視します。しかし、-pti*directory* フラグはソースファイルの検索規則を複雑にするので、-pti*directory* オプションの代わりに -I オプションを使用してください。

7.5.3 問題がある検索の回避

コンパイラがコンパイル対象ではないファイルを検索するために、紛らわしい警告あるいはエラーメッセージが生成されることがあります。通常、問題は、たとえば foo.h というファイルにテンプレート宣言が含まれていて、foo.cc などの別のファイルが暗黙で取り込まれることにあります。

ヘッダーファイル foo.h の中にテンプレート宣言が存在する場合は、コンパイラはデフォルトで、foo という名前および C++ のファイル拡張子 (.C、.c、.cc、.cpp、.cxx、または .c++) を持つファイルをデフォルトで検索します。そうしたファイルを見つけた場合、コンパイラはそのファイルを自動的に取り込みます。こうした検索の詳細は、105 ページの「7.5 テンプレート定義の検索」を参照してください。

このように扱われるべきでないファイル foo.cc が存在する場合、選択肢は 2 つあります。

- .h または .cc の名前を変更して、名前が一致しないようにする。
- -template=no%extdef オプションを指定することによって、テンプレート定義ファイルの自動検索を無効にする。この場合は、すべてのテンプレート定義をコードに明示的に取り込む必要があります。このため、「定義分離」モデルは使用できなくなります。

◆ ◆ ◆ 第 8 章

8 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。118 ページの「10.2 マルチスレッドプログラムでの例外の使用」にも補足情報を掲載しています。例外処理の詳細については、『プログラミング言語 C++』(第 3 版、Bjarne Stroustrup 著、アスキー、1997 年) を参照してください。

8.1 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例外をスローすることはできません。

8.2 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、前述の 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。

8.3 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。

8.4 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++ 標準で指定されるクラスと例外関連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。次は、`<exception>` ヘッダーファイル宣言を抜粋したものです。

```
// standard header <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 {...};
    // Unexpected exception handling
       typedef void (*unexpected_handler)();
       unexpected_handler
         set_unexpected(unexpected_handler) throw();
       void unexpected();
    // Termination handling
       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++ release 4.2 で使用される例外との互換性について、ヘッダー <exception.h> は標準モードでの使用のため提供されます。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++ 標準に従って更新してください。

```
// header <exception.h>, used for transition
#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++ release 4.2 で提供されているヘッダーと同じですので、ここでは再生されません。

8.5 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 (または catch ブロック) への、または try ブロック (または catch 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。

8.6 例外のある共有ライブラリの構築

C++ コードを含むプログラムは -Bsymbolic を使用せず、代わりにリンカーマップファイルまたはリンカースコープオプションを使用してください (65 ページの「4.1 リンカースコープ」 を参照)。-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。

◆ ◆ ◆ 第 9 章

9 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。全般的、および C++ のソフトウェアパフォーマンスに関して、多くの書物が制作されてきたため、この章ではそのような重要な情報を繰り返さず、C++ コンパイラに強く影響するパフォーマンスの手法のみを説明します。

9.1 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かぎりません。

プログラムの明瞭さを保ちつつ、一時オブジェクトの数が最少になるように関数を記述してください。このための手法としては、暗黙の一時オブジェクトに代わって明示的な変数を使用すること、値パラメータに代わって参照パラメータを使用することなどがあります。また、+ と = だけを実装して使用するのではなく、+= のような演算を実装および使用することもよい手法です。たとえば、次の例の最初の行は、a + b の結果に一時オブジェクトを使用していますが、2 行目は一時オブジェクトを使用していません。

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

9.2 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のに負荷が大きい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。

9.3 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前述し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。

9.4 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。クラスを直接渡されたオブジェクト。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、ポインタが戻ったあと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。

9.4.1 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。

- 可能なかぎ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。
- 可能なかぎ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。

9.4.2 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、アーキテクチャーによって異なります。

表 9–1 アーキテクチャー別の構造体と共用体の渡し方

アーキテクチャー	内容の説明
SPARC V7 および V8	構造体と共用体は、呼び出し元で記憶領域を割り当て、その記憶領域へのポインタを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。共用体およびそのほかのすべての構造体は、呼び出し元で記憶領域を割り当て、その記憶領域へのポインタを渡すことによって渡されます。つまり、小さな構造体はレジスタで渡され、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。
x86 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。

9.5 メンバー変数のキャッシュ

C++ メンバー関数では、メンバー変数へのアクセスが頻繁に行われます。

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

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

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

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

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

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

前述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 10 章

10 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。

マルチスレッド化の詳細は、『Multithreaded Programming Guide』を参照してください。

OpenMP 共有メモリー並列化指令を使用してマルチスレッドプログラムを作成する方法の詳細は、『OpenMP API ユーザガイド』も参照してください。

10.1 マルチスレッドプログラムの構築

C++ コンパイラに付属しているライブラリは、すべてマルチスレッドで使用しても安全です。マルチスレッドアプリケーションを作成したい場合や、アプリケーションをマルチスレッド化されたライブラリにリンクしたい場合は、`-mt` オプションを付けてプログラムのコンパイルとリンクを行う必要があります。このオプションを付けると、`–D_REENTRANT` がプリプロセッサに渡され、`–lthread` が `ld` に正しい順番で渡されます。互換性モード (`-compat[=4]`) の場合、`-mt` オプションは `libthread` を `libC` の前にリンクします。標準モード (デフォルトモード) の場合、`-mt` オプションは `libthread` を `ibCrun` の前にリンクします。マクロとライブラリを指定する代わりに、より簡単でエラーの発生しにくい方法である `–mt` を使用することをお勧めします。

10.1.1 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
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

10.1.2 C++ サポートライブラリの使用

C++ サポートライブラリ (libCrun、 ibiostream、libCstd、libC) は、マルチスレッドで使用しても安全ですが、非同期では安全 (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

10.2 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() または unexpected() 関数を設定できます。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、すべてのスレッドで abort() になります。107 ページの「8.2 実行時エラーの指定」 を参照してください。

10.2.1 スレッドの取り消し

-noex または -features=no%except、コンパイラオプションが指定されている場合を除き、pthread_cancel(3T) - の呼び出しでスレッドを取り消すと、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。

pthread_cancel(3T) では、例外と同じ仕組みが使用されます。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。

10.3 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 -library=Cstd`) は、いくつかのロケールを除けばマルチスレッドで使用しても安全なライブラリで、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することが保証されています。ただし、依然、スレッド間で共有するライブラリオブジェクト周りには注意を払う必要があります。`setlocale`(3C) および `attributes`(5) のマニュアルページを参照してください。

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>

//
// An integer shared among multiple threads.
//
int I;

//
// A mutex used to synchronize updates to I.
//
_RWSTDMutex I_mutex;

//
// Increment I by one. Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire(); // Lock the mutex.
   I++;
   I_mutex.release(); // Unlock the mutex.
}

//
// Decrement I by one. Uses an _RWSTDGuard.
//
```

```
void decrement_I ()
{
   _RWSTDGuard guard(I_mutex); // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

10.4 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、すなわちサポートされている Solaris オペレーティングシステムのバージョンで実行される、マルチスレッド機能を使用するプログラムから使用できます。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 ライブラリから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。

10.4.1 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、128 ページの「10.4.2 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 10–1 に示します。

表 10–1 iostream の中核クラス

クラス	内容の説明
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、そのあと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。

10.4.1.1 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。これらの関数を次に示します。

表 10–2 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	内容の説明

`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ーの先頭を指すとはかぎりません。

`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ーの先頭を指すとはかぎりません。

`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ーの先頭を指すとはかぎりません。

`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ーの先頭を指すとはかぎりません。

表 10–2 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	内容の説明
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します。ユーザーが用意したバッファーに、変換後の文字列を収容できるだけの大きさがなければいけません。

注 – 以前の libC との互換性を確保するために提供されている iostream ライブラリの公開変換ルーチン (oct、hex、dec、chr、form) は、マルチスレッドで使用すると安全ではありません。

10.4.1.2 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ーオプションを特に必要としません。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。

10.4.1.3 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。

例 10–1 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 istr。

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。

例 10–2 gcount の呼び出し

```
#include <iostream.h>
#include <rlocks.h>
void fetch_line(istream& istr, char* line, int& linecount)
{
    stream_locker sl(istr, stream_locker::lock_defer);

    sl.lock(); // lock the stream istr
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // unlock istr
    ...
}
```

この例では、stream_locker クラスの lock メンバー関数を呼び出してから unlock メンバー関数を呼び出すまでが、プログラムの相互排他領域になります。

ユーザー定義の入出力操作

マルチスレッドでの安全性を実現するには、別々の操作を特定の順番で行う必要があるユーザー定義型用の入出力操作を、危険領域としてロックする必要があります。この入出力操作の例を次に示します。

例 10–3 ユーザー定義の入出力操作

```
#include <rlocks.h>
#include <iostream.h>
class mystream: public istream {

    // other definitions...
    int getRecord(char* name, int& id, float& gpa);
};

int mystream::getRecord(char* name, int& id, float& gpa)
{
    stream_locker sl(this, stream_locker::lock_now);

    *this >> name;
    *this >> id;
    *this >> gpa;

    return this->fail() == 0;
}
```

10.4.1.4 マルチスレッドで使用しても安全なクラスのパフォーマンスオーバーヘッドの削減

現行の libC ライブラリに含まれているマルチスレッドで使用しても安全なクラスを使用すると、シングルスレッドアプリケーションの場合でさえも多少のオーバーヘッドが発生します。libC の unsafe_ クラスを使用すると、このオーバーヘッドを回避できます。

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。

例 10–4 マルチスレッドでの安全性の無効化

```
#include <iostream.h>
//disable mt-safety
cout.set_safe_flag(stream_MT::unsafe_object);
//disable mt-safety
cin.set_safe_flag(stream_MT::unsafe_object);
```

例 10–4 マルチスレッドでの安全性の無効化 (続き)

```
cout.put("4');
cin.read(buf, len);
```

iostream オブジェクトがマルチスレッドで使用しても安全な場合は、相互排他ロックを行うことで、そのオブジェクトのメンバー変数が保護されます。しかし、シングルスレッド環境でしか実行されないアプリケーションでは、このロック処理のために、本来なら必要のないオーバーヘッドがかかります。iostream オブジェクトのマルチスレッドでの安全性の有効/無効を動的に切り替えると、パフォーマンスを改善できます。たとえば、iostream オブジェクトのマルチスレッドでの安全性を無効にするには、次のようにします。

例 10–5 マルチスレッドでの安全性の無効化

```
fs.set_safe_flag(stream_MT::unsafe_object);// disable MT-safety
    .... do various i/o operations
```

iostream が複数のスレッド間で共有されないコード領域では、マルチスレッドでの安全性の無効化ストリームであっても、安全に使用できます。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。

例 10–6 マルチスレッドで使用すると安全ではないオブジェクトの同期処理

```
    generic_lock();
    fs.set_safe_flag(stream_MT::unsafe_object);
    ... do various i/o operations
    generic_unlock();
```

ここで、generic_lock 関数と generic_unlock 関数は、相互排他ロック (mutex)、セマフォー、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 – libC クラスによって提供される stream_locker クラスは、この目的で優先される機構です。

詳細は 131 ページの「10.4.5 オブジェクトのロック」を参照してください。

10.4.2 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。

10.4.2.1 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。

例 10–7 新しいクラス

```
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase
```

10.4.2.2 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。

例 10–8 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 {...};
```

10.4.2.3 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。

例 10–9 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();
```

例 10–9 新しい関数 *(続き)*

```
  int in_avail_unlocked();
  int sputbackc_unlocked(char);
  int sputc_unlocked(int);
  int sputn_unlocked(const char *, int);
  int out_waiting_unlocked();
protected:
  char* base_unlocked();
  char* ebuf_unlocked();
  int blen_unlocked();
  char* pbase_unlocked();
  char* eback_unlocked();
  char* gptr_unlocked();
  char* egptr_unlocked();
  char* pptr_unlocked();
  void setp_unlocked(char*, char*);
  void setg_unlocked(char*, char*, char*);
  void pbump_unlocked(int);
  void gbump_unlocked(int);
  void setb_unlocked(char*, char*, int);
  int unbuffered_unlocked();
  char *epptr_unlocked();
  void unbuffered_unlocked(int);
  int allocate_unlocked(int);
};

class filebuf: public streambuf {
public:
 int is_open_unlocked();
 filebuf* close_unlocked();
 filebuf* open_unlocked(const char*, int, int =
   filebuf::openprot);

 filebuf* attach_unlocked(int);
};

class strstreambuf: public streambuf {
public:
 int freeze_unlocked();
 char* str_unlocked();
};


unsafe_ostream& endl(unsafe_ostream&);
unsafe_ostream& ends(unsafe_ostream&);
unsafe_ostream& flush(unsafe_ostream&);
unsafe_istream& ws(unsafe_istream&);
unsafe_ios& dec(unsafe_ios&);
unsafe_ios& hex(unsafe_ios&);
unsafe_ios& oct(unsafe_ios&);

char* dec_r (char* buf, int buflen, long num, int width)
char* hex_r (char* buf, int buflen, long num, int width)
char* oct_r (char* buf, int buflen, long num, int width)
char* chr_r (char* buf, int buflen, long chr, int width)
char* str_r (char* buf, int buflen, const char* format, int width
   = 0);
```

例 10–9 新しい関数 *(続き)*

```
char* form_r (char* buf, int buflen, const char* format,...)
```

10.4.3 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ほか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、ほか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、ほか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。たとえば、次を見てください。

```
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、`libC` ライブラリはシーケンスのその次の文字を返すことを保証しません。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、131 ページの「10.4.5 オブジェクトのロック」を参照してください。

10.4.4 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。次は

```
cout << " Error message:" << errstring[err_number] << "\n";
```

このコードでは、cout ストリームオブジェクトの 3 つのメンバー関数が実行されます。cout は共有オブジェクトなので、マルチスレッド環境では、この操作全体を危険領域として不可分的に (連続して) 実行しなければなりません。iostream クラスのオブジェクトに対する一続きの操作を不可分的に実行するには、何らかのロック処理が必要です。

iostream オブジェクトをロックできるように、libC ライブラリに新しく stream_locker クラスが追加されています。stream_locker クラスの詳細については、131 ページの「10.4.5 オブジェクトのロック」を参照してください。

10.4.5 オブジェクトのロック

共有オブジェクトに対処する方法とマルチスレッド化は 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。たとえば、次を見てください。

- 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。

10.4.5.1 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。

例 10–10 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);
    .....// open file
    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、そのあと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。

例 10–11 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();  // lock openfile_stream
    openfile_stream << "Value: " << int_value << "\n";
    if(!openfile_stream) {
            file_error("Output of value failed\n");
```

例 10–11　入出力操作とエラーチェックの不可分化　　(続き)

```
            return;
    }
    file_lck.unlock(); // unlock openfile_stream
}
```

詳細は、stream_locker(3CC4) のマニュアルページを参照してください。

10.4.6　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、次のとおりです: xsgetn、underflow、pbackfail、xsputn、overflow、seekoff、seekpos。
- 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。

10.4.7 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次に示します。

例 10–12 共有オブジェクトの破棄

```
#include <fstream.h>
#include <thread.h>
fstream* fp;

void *process_rtn(void*)
{
    // body of sub-threads which uses fp...
}

void multi_process(const char* filename, int numthreads)
{
    fp = new fstream(filename, ios::in); // create fstream object
                                         // before creating threads.
    // create threads
    for (int i=0; i<numthreads; i++)
            thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);

        ...
    // wait for threads to finish
    for (int i=0; i<numthreads; i++)
            thr_join(0, 0, 0);

    delete fp;                          // delete fstream object after
    fp = NULL;                         // all threads have completed.
}
```

10.4.8 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。

例 10–13 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

```
// create tagged thread data
// the output file is of the form:
//         <tag><string of data>\n
// where tag is an integer value in a unsigned char.
// Allows up to 255 threads to be run in this application
```

例 10–13 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)

```
// <string of data> is any printable characters
// Because tag is an integer value written as char,
// you need to use od to look at the output file, suggest:
//            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;

// entry routine for each thread
void* ThreadDuties(void* v) {
// obtain arguments for this thread
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // open thread input file
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // read a line at a time
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // lock cout stream so the i/o operation is atomic
    lockout.lock();
  // tag line and send to cout
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  // argv: 1+ list of filenames per thread
   if(argc < 2) {
     cout << "usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// array of thread_ids
  thread_t created_threads[thread_bufsize];
// array of arguments to thread entry routine
  thread_args thr_args[thread_bufsize];
  int i;
  for(i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
```

例 10–13 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)

```
// assign a tag to a thread - a value less than 256
    thr_args[i].thread_tag = total_tags++;
// create threads
    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;
}
```

10.5 メモリーバリアー組み込み関数

コンパイラには、SPARC プロセッサと x86 プロセッサ用のさまざまなメモリーバリアー組み込み関数を定義するヘッダーファイル `mbarrier.h` が用意されています。これらの組み込み関数は、開発者が独自の同期プリミティブを使用してマルチスレッドコードを記述するために使用できます。これらの組み込み関数がいつ必要になるか、また、特定の状況で必要かどうかを判断するために、ユーザーは使用しているプロセッサのドキュメントを参照することが推奨されます。

`mbarrier.h` によりサポートされるメモリーオーダリング組み込み関数

- `__machine_r_barrier()` — これは、*read* バリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロード操作の前に完了します。
- `__machine_w_barrier()` — これは、*write* バリアーです。これにより、バリアー前のすべての格納操作が、バリアー後のすべての格納操作の前に完了します。
- `__machine_rw_barrier()` — これは、*read—write* バリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- `__machine_acq_barrier()` — これは、*acquire* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- `__machine_rel_barrier()` — これは、*release* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべての格納操作の前に完了します。
- `__compiler_barrier()` — コンパイラが、バリアーを越えてメモリーアクセスを移動しないようにします。

__compiler_barrier() 組み込み関数を除くすべてのバリアー組み込み関数は、メモリーオーダリング組み込み関数を生成します。x86 では、mfence、sfence、または lfence 組み込み関数で、SPARC プラットフォームでは membar 組み込み関数です。

__compiler_barrier() 組み込み関数は、命令を生成せず、代わりに今後メモリー操作を開始する前にそれまでのメモリー操作をすべて完了する必要があることをコンパイラに通知します。この実際の結果として、ローカルでないすべての変数、および static 記憶クラス指定子を持つローカル変数が、バリアー前のメモリーに再度格納されてバリアー後に再ロードされるため、コンパイラではバリアー前のメモリー操作とバリアー後のメモリー操作が混在することはありません。ほかのすべてのバリアーには、__compiler_barrier() 組み込み関数の動作が暗黙的に含まれています。

たとえば、次のコードでは、__compiler_barrier() 組み込み関数が存在しているためコンパイラによる 2 つのループのマージが止まります。

```
#include "mbarrier.h"
int thread_start[16];
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
      while (thread_start[i]==1){}
   }
}
```

パート III

ライブラリ

◆ ◆ ◆ 第 11 章

11 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。

11.1 C ライブラリ

Oracle Solaris オペレーティングシステムでは、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` および `libm` 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。

11.2 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 11–1　C++ コンパイラに添付されるライブラリ

ライブラリ	内容の説明	使用できるモード
libstlport	標準ライブラリの STLport 実装	-compat=5
libstlport_dbg	デバッグモード用 STLport ライブラリ	-compat=5
libCrun	C++ 実行時	-compat=5
libCstd	C++ 標準ライブラリ	-compat=5
libiostream	従来の iostream	-compat=5
libC	C++ 実行時、従来の iostream	-compat=4
libcsunimath	-xia オプションをサポート	-compat=5
libcomplex	複素数ライブラリ	-compat=4
librwtool	Tools.h++ 7	-compat=4、 -compat=5
librwtool_dbg	デバッグ可能な Tools.h++ 7	-compat=4,-compat=5
libgc	ガベージコレクション	C インタフェース
libdemangle	復号化	C インタフェース

注 – STLport、Rogue Wave、または Solaris Studio C++ ライブラリの構成マクロを再定義したり変更したりしないでください。ライブラリは C++ コンパイラとともに動作するよう構成および構築されています。libCstd と Tool.h++ は互いに働き合うように構成されているので、その構成マクロを変更すると、プログラムのコンパイルやリンクが行われなくなったり、プログラムが正しく実行されなくなったりします。

11.2.1 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。

- `libiostream`: これは標準モード (`-compat=5`) で構築した従来の iostream ライブラリです。既存のソースで従来の iostream を使用している場合には、`libiostream` を使用すれば、ソースを修正しなくてもこれらのソースを標準モード (`-compat=5`) でコンパイルできます。このライブラリを使用するには、`-library=iostream` を使用します。

注 – 標準ライブラリのほとんどの部分は、標準 iostream を使用することに依存しています。従来の iostream を同一のプログラム内で使用すると、問題が発生する可能性があります。

- `libC`: これは互換モード (`-compat=4`) で必要なライブラリです。このライブラリには C++ 実行時サポートだけでなく従来の iostream も含まれています。
- `libcomplex`: このライブラリは、互換モード (`-compat=4`) で複素数の演算を行うときに必要です。標準モードの場合は、`libCstd` の複素数演算の機能が使用されます。
- `libstlport`: これは、C++ 標準ライブラリの STLport 実装です。このライブラリを使用するには、デフォルトの `libCstd` の代わりにオプション `-library=stlport4` を指定します。ただし、`libstlport` と `libCstd` の両方を同一プログラム内で使用することはできません。インポートしたライブラリを含み、すべてをどちらか一方のライブラリだけを使ってコンパイルしリンクする必要があります。
- `librwtool` (`Tools.h++`): Tools.h++ は、RogueWave の C++ 基礎クラスライブラリです。Version 7 が提供されています。このライブラリは廃止され、新しいコードでこのライブラリを使用することは非推奨です。RW Tools.h++ を使用していた、C++ 4.2 用に作成されたプログラムに対応する目的で、このライブラリは提供されています。
- `libgc`: このライブラリは、展開モードまたはガ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できます。プログラムを `libgc` ライブラリとリンクする場合は、`free` や `delete` を呼び出さずに、それ以外は通常どおりにプログラムを記述できます。ガベージコレクションライブラリは、動的読み込みライブラリと依存関係があるため、プログラムのリンクでは、`-lgc` および `-ldl` を指定します。

  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。

11.2.2 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 1、3、3C++、および 3cc4 の各節にあります。

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 Version 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
```

11.2.3 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 -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 -lc
```

詳細は、257 ページの「A.2.49 -library=*l*[,*l*...]」を参照してください。

11.3 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 または C99 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。
- Solaris Studio C++ コンパイラに添付された次のライブラリを指定するには、-library オプションを使用します。

- libCrun
- libCstd
- libiostream
- libC
- libcomplex
- libstlport、libstlport_dbg
- librwtool、librwtool_dbg
- libgc

注 – librwtool の従来の iostream 形式を使用するには、-library=rwtools7 オプションを使用します。librwtool の標準 iostream 形式を使用するには、-library=rwtools7_std オプションを使用します。

-library オプションと -staticlib オプションの両方に指定されたライブラリは静的にリンクされます。次にその例を挙げます。

- libstdcxx (Solaris OS の一部として配布)

次のコマンドでは Tools.h++ Version 7 の従来の iostream 形式と libiostream ライブラリが動的にリンクされます。

```
example% CC test.cc -library=rwtools7,iostream
```

- 次のコマンドでは libgc ライブラリが静的にリンクされます。

```
example% CC test.cc -library=gc -staticlib=gc
```

- 次のコマンドでは test.cc が互換モードでコンパイルされ、libC が静的にリンクされます。互換モードでは libC がデフォルトでリンクされるので、このライブラリを -library オプションで指定する必要はありません。

```
example% CC test.cc -compat=4 -staticlib=libC
```

- 次のコマンドでは ライブラリ libCrunおよび libCstd がリンク対象から除外されます。指定しない場合は、これらのライブラリは自動的にリンクされます。

```
example% CC test.cc -library=no%Crun,no%Cstd
```

デフォルトでは、CC は、指定されたコマンド行オプションに従ってさまざなシステムライブラリをリンクします。-xnolib (または -nolib) を指定した場合、CC は、-l オプションを使用してコマンド行で明示的に指定したライブラリだけをリンクします。-xnolib または -nolib を使用した場合、-library オプションが存在していても無視されます。

-R オプションは、動的ライブラリの検索パスを実行可能ファイルに組み込むときに使用します。実行時リンカーは、実行時にこれらのパスを使ってアプリケーションに必要な共有ライブラリを探します。CC ドライバは、デフォルトで -R*install_directory*/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。

デフォルトでは、リンカーは /lib および /usr/lib を検索します。-L オプションでこれらのディレクトリやコンパイラのインストールディレクトリを指定しないでください。

配備用に構築するプログラムは、コンパイラのディレクトリでライブラリを参照することを防止する -norunpath または -R オプションを使用して構築するべきです (150 ページの「11.6 共有ライブラリの使用」を参照してください)。

11.4 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。

11.4.1 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。このライブラリに標準はありません。互換モードの libC の一部であり、標準モードの libiostream にもあります。
- 標準の **iostream**。これは C++ 標準ライブラリ libCstd に含まれていて、標準モードだけで使用されます。これは、バイナリレベルでもソースレベルでも「従来の iostream」とは互換性がありません。

すでに C++ のソースがある場合、そのコードは従来の iostream を使用しており、次の例のような形式になっていると思われます。

```
// file prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

次のコマンドは、互換性モードで prog1.cc をコンパイル、リンクして、prog1 という実行可能なプログラムを生成します。従来の iostream ライブラリは、互換性モードのときにデフォルトでリンクされる libC ライブラリに含まれています。

```
example% CC -compat prog1.cc -o prog1
```

次の例では、標準 iostream が使用されています。

```
// file prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

次のコマンドは、prog2.cc をコンパイル、リンクして、prog2 という実行可能なプログラムを生成します。コンパイルは標準モードで行われ、このモードでは、標準の iostream ライブラリを含む libCstd がデフォルトでリンクされます。

```
example% CC prog2.cc -o prog2
```

コンパイルモードの詳しい説明については、『C++ 移行ガイド』を参照してください。

11.4.2 complex ライブラリ

標準ライブラリには、C++ 4.2 コンパイラに付属していた complex ライブラリに似た、テンプレート化された complex ライブラリがあります。標準モードでコンパイルする場合は、<complex.h> ではなく、<complex> を使用する必要があります。互換性モードで <complex> を使用することはできません。

互換性モードでは、リンク時に complex ライブラリを明示的に指定しなければなりません。標準モードでは、complex ライブラリは libCstd に含まれており、デフォルトでリンクされます。

標準モード用の complex.h ヘッダーはありません。C++ 4.2 では、「complex」 はクラス名ですが、標準 C++ では「complex」はテンプレート名です。したがって、旧式のコードを変更せずに動作できるようにする typedef を使用することはできません。このため、複素数を使用する、4.2 用のコードで標準ライブラリを使用するには、多少の編集が必要になります。たとえば、次のコードは 4.2 用に作成されたものであり、互換性モードでコンパイルされます。

```
// file ex1.cc (compatibility mode)
#include <iostream.h>
#include <complex.h>

int main()
```

```
{
    complex x(3,3), y(4,4);
    complex z = x * y;
    cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
}
```

次の例では、ex1.cc を互換モードでコンパイル、リンクし、生成されたプログラムを実行しています。

```
example% CC -compat ex1.cc -library=complex
example% a.out
x=(3, 3), y=(4, 4), z=(0, 24)
```

次は、標準モードでコンパイルされるように ex2.cc と書き直された ex1.cc です。

```
// file ex2.cc (ex1.cc rewritten for standard mode)
#include <iostream>
#include <complex>
using std::complex;

int main()
{
     complex<double> x(3,3), y(4,4);
     complex<double> z = x * y;
     std::cout << "x=" << x << ", y=" << y << ", z=" << z <<
       std::endl;
}
```

次の例では、書き直された ex2.cc をコンパイル、リンクして、生成されたプログラムを実行しています。

```
% CC ex2.cc
% a.out
x=(3,3), y=(4,4), z=(0,24)
```

複素数演算ライブラリの使用方法についての詳細は、表 13–4 を参照してください。

11.4.3 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、257 ページの「A.2.49 -library=*l*[,*l*...]」を参照してください。

表 11–2 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

表 11–2 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
Tools.h++ Version 7	-compat=4 -compat=5	-library=rwtools7 -library=rwtools7,iostream -library=rwtools7_std
デバッグ対応 Tools.h++ Version 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
STLport Version 4	-compat=5	-library=stlport4
STLport Version 4 デバッグ	-compat=5	-library=stlport4_dbg
Apache stdcxx バージョン 4	-compat=5	-library=stdcxx4

11.5 標準ライブラリの静的リンク

デフォルト時、CC ドライバは、デフォルトライブラリの -l*lib* オプションをリンカーに渡すことによって、libc と libm を含むいくつかのライブラリの共有バージョンでリンクします。互換性モードと標準モードにおけるデフォルトライブラリのリストについては、144 ページの「11.2.3 デフォルトの C++ ライブラリ」を参照してください。

このようにデフォルトのライブラリを静的にリンクする場合、-library オプションと -staticlib オプションを一緒に使用すれば、C++ ライブラリを静的にリンクできます。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。次の例は、Solaris 8 または Solaris 9 オペレーティングシステムで libCrun と静的に、libm および 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 -lc
```

-l オプションの順序は重要です。-lCstd、-lCrun、および -lm オプションは、-lc の前に表示します。

注 – libCrun および libCstd を静的にリンクすることはお勧めできません。/usr/lib 内の動的バージョンは、インストール先の Solaris のバージョンで動作するよう構築します。

ほか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、329 ページの「A.2.153 -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。

注 – /lib および /usr/lib にある Solaris ライブラリの静的バージョンは、もう使用できません。たとえば、libc を静的にリンクしようとする試みは、失敗します。

```
CC hello.cc -xnolib -lCrun -lCstd -Bstatic -lc
```

11.6 共有ライブラリの使用

次の C++ 実行時共有ライブラリは、C++ コンパイラの一部として出荷されています。

- libCCexcept.so.1 (SPARC Solaris のみ)
- libcomplex.so.5 (Solaris のみ)
- librwtool.so.2
- libstlport.so.1

Linux では、次の追加ライブラリが C++ コンパイラの一部として出荷されています。

- libCrun.so.1
- libCstd.so.1
- libdemangle.so
- libiostream.so.1

Solaris 10 では、次の追加ライブラリがほかのライブラリとともに、C++ 実行時ライブラリパッケージである SUNWlibC の一部としてインストールされます。

アプリケーションが、C++ コンパイラの一部として出荷されている共有ライブラリのいずれかを使用している場合は、CC ドライバは *runpath* に調整を加え (-R オプションを参照)、実行可能ファイルの構築に使用するライブラリの場所を指すように

します。あとで、同じバージョンのコンパイラを同じ場所にインストールしていないコンピュータに実行可能ファイルを配備する場合は、必要な共有ライブラリが見つかりません。

プログラムの起動時に、ライブラリはまったく見つからない、あるいは誤ったバージョンのライブラリが使用される可能性があり、プログラムの正しくない動作につながります。このような状況では、必要なライブラリをプログラムと共に出荷し、それらのライブラリのインストール場所を指す *runpath* を指定して構築を行うべきです。

『Using and Redistributing Solaris Studio Libraries in an Application』という資料には、このトピックに関する詳細な説明と例が掲載されています。これは、`http://developers.sun.com/sunstudio/documentation/techart/stdlibdistr.html` から入手できます。

11.7 C++ 標準ライブラリの置き換え

ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。基本的な操作としては、コンパイラに添付されている標準のヘッダーとライブラリを無効にして、新しいヘッダーファイルとライブラリが格納されているディレクトリとライブラリ自身の名前を指定します。

コンパイラでは、標準ライブラリの STL ポートおよび Apache `stdcxx` 実装がサポートされます。詳細は、173 ページの「12.3 STLport」と 174 ページの「12.4 Apache stdcxx 標準ライブラリ」を参照してください。

11.7.1 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。置き換えるライブラリが `libCstd` である場合は、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。

11.7.2 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 オペレーティングシステムと基本 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 バージョン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。

11.7.3 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。

11.7.4 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... (compile)
```

-library=no%Cstd オプションを指定しているため、コンパイル中、コンパイラ独自のバージョンのヘッダーがインストールされているディレクトリは検索されません。

プログラムまたはライブラリのリンクごとに -library=no%Cstd オプションを指定して、コンパイラ独自の libCstd が検出されないようにします。さらに、-L オプションを指定して、代替ライブラリがインストールされているディレクトリを指示します。さらに、-l オプションを指定して、代替ライブラリを指定します。次に例を示します。

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd... (link)
```

あるいは、-L や -l オプションを使用せずに、ライブラリの絶対パス名を直接指定することもできます。次に例を示します。

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a... (link)
```

-library=no%Cstd オプションを指定しているため、リンク中、コンパイラ独自のバージョンの libCstd はリンクされません。

11.7.5 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 オペレーティングシステム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。Version 8 より前のリリースの Solaris オペレーティングシステムの 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。

- ヘッダーファイルにファイル名接尾辞 (拡張子) がない場合に、ヘッダーファイルのみを検索すること、またはヘッダーファイルに関する規則を示す makefile を作成することができない。
- string というディレクトリまたは実行可能プログラムがあると、そのディレクトリまたはプログラムが標準ヘッダーファイルの代わりに検出される可能性がある。
- Solaris 8 オペレーティングシステムより前のリリースの Solaris オペレーティングシステムでは .KEEP_STATE が有効なときのメイクファイルのデフォルトの相互依存関係により、標準ヘッダーが実行可能プログラムに置き換えられる可能性がある (デフォルトの場合、接尾辞がないファイルは構築対象プログラムとみなされる)。

こうした問題を解決するため、コンパイラの include ディレクトリには、ヘッダーと同じ名前を持つファイルと、 一意の接尾辞 .SUNWCCh を持つ、そのファイルへのシンボリックリンクが含まれています。 SUNW はコンパイラに関係するあらゆるパッケージに対する接頭辞、CC は C++ コンパイラの意味、.h はヘッダーファイルの

通常の接尾辞です。つまり <string> と指定された場合、コンパイラは <string.SUNWCCh> と書き換え、その名前を検索します。接尾辞付きの名前は、コンパイラ専用の include ディレクトリにだけ存在します。このようにして見つけられたファイルがシンボリックリンクの場合 (通常はそうである)、コンパイラは、エラーメッセージやデバッガの参照でそのリンクを 1 回だけ間接参照し、その参照結果 (この場合は string) をファイル名として使用します。ファイルの依存関係情報を送るときは、接尾辞付きの名前の方が使用されます。

この名前の書き換えは、2 つのバージョンがある 17 個の標準 C ヘッダーと 32 個の標準 C++ ヘッダーのいずれかを、パスを指定せずに山括弧 < > で囲んで指定した場合にだけ行われます。山括弧の代わりに引用符が使用されるか、パスが指定されるか、ほか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 11–3　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	山括弧ではなく、二重引用符
<../string>	../string	パス指定がある場合

コンパイラが *header*.SUNWCCh (header はヘッダー名) を見つけられない、コンパイラは、#include 指令で指定された名前で検索し直します。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。

11.7.5.1 標準 C++ ヘッダーの置き換え

153 ページの「11.7.5 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、152 ページの「11.7.3 代替ライブラリのインストール」で説明している SUNWCCh バージョンの代替 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 という名前のほか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。

11.7.5.2 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> の代替ヘッダーがあるとします。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 12 章

12 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 標準 C++ ライブラリ、Version 2 に基づいています。このライブラリは、コンパイラのデフォルトモード (`-compat=5`) でのみ使用でき、`-compat[=4]` オプションと組み合わせた使用はサポートされていません。

また、C++ コンパイラで、STLport の標準ライブラリの Version 4.5.3 がサポートされました。libCstd がデフォルトのライブラリですが、代わりに STLport の製品を使用できるようになりました。詳細は、173 ページの「12.3 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できます。その場合は、-library=no%Cstd オプションを使用します。ただし、コンパイラに添付された標準ライブラリを置き換えることは危険で、必ずしもよい結果につながるわけではありません。詳細は、151 ページの「11.7 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。

12.1 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は表 12–1 に一覧表示します。

表 12–1 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	内容の説明
<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
<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

表 12–1　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	内容の説明
<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

12.2　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ドキュメントページを表 12–2 に一覧表示します。

表 12–2　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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
Containers	標準テンプレートライブラリ (STL) コレクション
Forward_Iterators	読み書きの両方が可能な順方向反復子
Function_Objects	operator() が定義済みのオブジェクト
Heap_Operations	make_heap、pop_heap、push_heap、sort_heap を参照
Input_Iterators	読み取り専用の順方向反復子
Insert_Iterators	反復子がコンテナ内の要素を上書きせずにコンテナに挿入することを可能にする、反復子アダプタ
Iterators	コレクションをたどったり、変更したりするためのポインタ汎用化機能
Negators	述語関数オブジェクトの意味を逆にするための関数アダプタと関数オブジェクト
Operators	C++ 標準テンプレートライブラリ出力用の演算子
Output_Iterators	書き込み専用の順方向反復子
Predicates	ブール値 (真偽) または整数値を返す関数または関数オブジェクト
Random_Access_Iterators	コンテナの読み取りと書き込みをして、コンテナにランダムアクセスすることを可能にする反復子
Sequences	一群のシーケンスをまとめたコンテナ
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

表 12–2　C++ 標準ライブラリのマニュアルページ　　(続き)

マニュアルページ	概要
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
basic_fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
basic_ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りをサポートする
basic_ios	すべてのストリームが共通に必要とする関数を取り込む基底クラス
basic_iostream	ストリームバッファーが制御する文字シーケンスの書式設定と解釈をサポートする
basic_istream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
basic_istringstream	メモリー上の配列から basic_string<charT,traits,Allocator> クラスのオブジェクトの読み取りをサポートする
basic_ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
basic_ostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
basic_ostringstream	basic_string<charT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
basic_streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
basic_string	文字に似た要素シーケンスを処理するためのテンプレート化されたクラス
basic_stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
basic_stringstream	メモリー上の配列への basic_string<charT,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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
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ー
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ー
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ー
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ー
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していないストリームバッファーに対する出力を制御する
char_traits	basic_string コンテナと iostream クラス用の型と演算を持つ特性 (traits) クラス
cin	<cstdio> で宣言されたオブジェクトの stdin に関連付けられたストリームバッファーからの入力を制御する
clog	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
codecvt	コード変換ファセット
codecvt_byname	指定ロケールに基づいたコードセット変換分類機能を含むファセット
collate	文字列照合、比較、ハッシュファセット
collate_byname	文字列照合、比較、ハッシュファセット
compare	真または偽を返す 2 項関数または関数オブジェクト
complex	C++ 複素数ライブラリ
copy	ある範囲の要素をコピーする
copy_backward	ある範囲の要素をコピーする
count	指定条件を満たすコンテナ内の要素の個数をカウントする

表 12–2　C++ 標準ライブラリのマニュアルページ　　(続き)

マニュアルページ	概要
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
cout	<cstdio> で宣言されたオブジェクトの stderr に関連付けられたストリームバッファーに対する出力を制御する
ctype	文字分類機能を取り込むファセット
ctype_byname	指定ロケールに基づいた文字分類機能を含むファセット
deque	ランダムアクセス反復子と、先頭および末尾の両方での効率的な挿入と削除をサポートするシーケンス
distance	2 つの反復子間の距離を求める
divides	1 つ目の引数を 2 つ目の引数で除算した結果を返す
equal	2 つのある範囲が等しいかどうか比較する
equal_range	並べ替えの順序を崩さずに値を挿入できる最大の二次範囲をコレクションから検出する
equal_to	1 つ目と 2 つ目の引数が等しい場合に真を返す 2 項関数オブジェクト
例外	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
fill	指定された値である範囲を初期化する
fill_n	指定された値である範囲を初期化する
find	シーケンスから値に一致するものを検出する
find_end	シーケンスからサブシーケンスに最後に一致するものを検出する
find_first_of	シーケンスから、別のシーケンスの任意の値に一致するものを検出する
find_if	シーケンスから指定された判定子を満たす値に一致するものを検出する
for_each	ある範囲のすべての要素に関数を適用する
fpos	iostream クラスの位置情報を保持する
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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
fstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
generate	値生成クラスによって生成された値でコンテナを初期化する
generate_n	値生成クラスによって生成された値でコンテナを初期化する
get_temporary_buffer	メモリーを処理するためのポインタベースのプリミティブ
greater	1 つ目の引数が 2 つ目の引数より大きい場合に真を返す 2 項関数オブジェクト
greater_equal	1 つ目の引数が 2 つ目の引数より大きいか等しい場合に真を返す 2 項関数オブジェクト
gslice	配列から汎用化されたスライスを表現するために使用される数値配列クラス
gslice_array	valarray から BLAS に似たスライスを表現するために使用される数値配列クラス
has_facet	ロケールに指定ファセットがあるかどうかを判定するための関数テンプレート
ifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りをサポートする
includes	ソートされたシーケンスに対する基本演算セット
indirect_array	valarray から選択された要素の表現に使用される数値配列クラス
inner_product	2 つの範囲 A および B の内積 (A × B) を求める
inplace_merge	ソートされた 2 つのシーケンスを 1 つにマージする
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共通に必要とする関数を取り込む基底クラス
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

表 12–2　C++ 標準ライブラリのマニュアルページ　　(続き)

マニュアルページ	概要
iosfwd	入出力ライブラリテンプレートクラスを宣言し、そのクラスを wide および tiny 型文字専用にする
isalnum	文字が英字または数字のどちらであるかを判定する
isalpha	文字が英字であるかどうかを判定する
iscntrl	文字が制御文字であるかどうかを判定する
isdigit	文字が 10 進数であるかどうかを判定する
isgraph	文字が図形文字であるかどうかを判定する
islower	文字が英小文字であるかどうかを判定する
isprint	文字が印刷可能かどうかを判定する
ispunct	文字が区切り文字であるかどうかを判定する
isspace	文字が空白文字であるかどうかを判定する
istream	ストリームバッファーが制御する文字シーケンスからの入力の読み取りと解釈をサポートする
istream_iterator	istream に対する反復子機能を持つストリーム反復子
istreambuf_iterator	作成元のストリームバッファーから連続する文字を読み取る
istringstream	メモリー上の配列からの basic_string<charT,traits,Alocator> 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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
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
ロケール	多相性を持つ複数のファセットからなるローカリゼーションクラス
logical_and	1 つ目の 2 つ目の引数が等しい場合に真を返す場合に 2 項関数オブジェクト
logical_not	引数が偽の場合に真を返す単項関数オブジェクト
logical_or	引数のいずれかが真の場合に真を返す 2 項関数オブジェクト
lower_bound	ソートされたコンテナ内の最初に有効な要素位置を求める
make_heap	ヒープを作成する
map	一意のキーを使用してキー以外の値にアクセスする連想コンテナ
mask_array	valarray の選別ビューを提供する数値配列クラス
max	2 つの値の大きい方の値を検出して返す
max_element	1 つの範囲内の最大値を検出する
mem_fun	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun1	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
mem_fun_ref	大域関数の代わりとしてポインタをメンバー関数に適合させる関数オブジェクト
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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
mismatch	2 つのシーケンスの要素を比較して、互いに値が一致しない最初の 2 つの要素を返す
modulus	1 つ目の引数を 2 つ目の引数で除算することによって得られた余りを返す
money_get	入力に対する通貨書式設定ファセット
money_put	出力に対する通貨書式設定ファセット
moneypunct	通貨句読文字ファセット
moneypunct_byname	通貨句読文字ファセット
multimap	キーを使用してコンテナキーでない値にアクセスするための連想コンテナ
multiplies	1 つ目と 2 つ目の引数を乗算した結果を返す 2 項関数オブジェクト
multiset	格納済みのキー値に高速アクセスするための連想コンテナ
negate	引数の否定値を返す単項関数オブジェクト
next_permutation	並べ替え関数に基づいてシーケンスの内容を連続的に入れ替えたものを生成する
not1	単項述語関数オブジェクトの意味を逆にするための関数アダプタ
not2	単項述語関数オブジェクトの意味を逆にするための関数アダプタ
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* 番目の要素よりあと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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
ofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
ostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
ostream_iterator	ostream と istream に反復子を使用可能にするストリーム反復子
ostreambuf_iterator	作成元のストリームバッファーに連続する文字を書き込む
ostringstream	basic_string<charT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
ostrstream	メモリー上の配列に書き込みを行う
pair	異種の値の組み合わせ用テンプレート
partial_sort	エンティティー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	エンティティー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ーを、満たさない全エンティティーの前に書き込む
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

表 12–2　C++ 標準ライブラリのマニュアルページ　　(続き)

マニュアルページ	概要
push_heap	ヒープに新しい要素を書き込む
queue	先入れ先出しの待ち行列のように振る舞うコンテナアダプタ
random_shuffle	コレクションの要素を無作為にシャッフルする
raw_storage_iterator	反復子ベースのアルゴリズムが初期化されていないメモリーに結果を書き込めるようにする
remove	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_copy	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_copy_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
remove_if	目的の要素をコンテナの先頭に移動し、目的の要素シーケンスの終了位置を表す反復子を返す
replace	コレクション内の要素の値を置換する
replace_copy	コレクション内の要素の値を置換して、置換後のシーケンスを結果に移動する
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

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
set	一意のキーを扱う連想コンテナ
set_difference	ソートされた差を作成する基本的な集合演算
set_intersection	ソートされた積集合を作成する基本的な集合演算
set_symmetric_difference	ソートされた対称差を作成する基本的な集合演算
set_union	ソートされた和集合を作成する基本的な集合演算
slice	配列の BLAS に似たスライスを表す数値配列クラス
slice_array	valarray の BLAS に似たスライスを表す数値配列クラス
smanip	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
smanip_fill	パラメータ化されたマニピュレータを実装するときに使用する補助クラス
sort	エンティティー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ーを満たさない全エンティティーの前に書き込む
stable_sort	エンティティーのコレクションをソートするためのテンプレート化されたアルゴリズム
stack	先入れ先出しのスタックのように振る舞うコンテナアダプタ
streambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
string	basic_string<char、char_traits<char>、allocator<char>> 用の型定義
stringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける
stringstream	メモリー上の配列上の basic_string<charT,traits,Alocator> 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 12–2　C++ 標準ライブラリのマニュアルページ　　(続き)

マニュアルページ	概要
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
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していないストリームバッファーに対する出力を制御する

表 12–2 C++ 標準ライブラリのマニュアルページ (続き)

マニュアルページ	概要
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	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスに対する読み書きをサポートする
wifstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスからの読み取りをサポートする
wios	すべてのストリームが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文字シーケンスからの入力の読み取りと解釈をサポートする
wistringstream	メモリー上の配列からの basic_string<charT,traits,Allocator&; クラスのオブジェクトの読み取りをサポートする
wofstream	1 つのファイル記述子に関連付けられた、複数の指定ファイルまたはその他デバイスへの書き込みをサポートする
wostream	ストリームバッファーが制御するシーケンスに対する出力の書式設定と書き込みをサポートする
wostringstream	basic_string<charT,traits,Allocator> クラスのオブジェクトの書き込みをサポートする
wstreambuf	各種のストリームバッファーを派生させて、文字シーケンスを制御しやすいようにする抽象基底クラス
wstring	basic_string<wchar_t, char_traits<wchar_t>、allocator<wchar_t>>用の型定義
wstringbuf	入力または出力シーケンスを任意の文字シーケンスに関連付ける

12.3 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

詳細は、257 ページの「A.2.49 -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、次のことを考慮して判断してください。

- STLport は、オープンソースの製品で、リリース間での互換性は保証されません。つまり、将来のバージョンの STLport でコンパイルすると、STLport 4.5.3 でコンパイルしたアプリケーションで問題が発生する可能性があります。また、STLport 4.5.3 でコンパイルしたバイナリは、将来のバージョンの STLport でコンパイルしたバイナリとリンクできない可能性があります。
- stlport4、Cstd、および iostream のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。
- コンパイラの将来のリリースには、STLport4 が含まれない可能性があります。STLport の新しいバージョンだけが含まれる可能性があります。コンパイラオプションの -library=stlport4 は、将来のリリースでは使用できず、STLport のそれ以降のバージョンを示すオプションに変更される可能性があります。
- STLport では、Tools.h++ はサポートされません。
- STLport は、デフォルトの libCstd とはバイナリ互換ではありません。STLport の標準ライブラリの実装を使用する場合は、-library=stlport4 オプションを指定してすべてのファイルのコンパイルおよびリンクを実行する必要があります。このことは、たとえば STLport 実装と C++ 区間演算ライブラリ libCsunimath を同時に使用できないことを意味します。その理由は、libCsunimath のコンパイルに使用されたのが、STLport ではなくデフォルトライブラリヘッダーであるためです。
- STLport 実装を使用する場合は、コードから暗黙に参照されるヘッダーファイルをインクルードしてください。標準のヘッダーは、実装の一部として相互にインクルードできます (必須ではありません)。
- -compat=4 によってコンパイルする場合には、STLport 実装を使用できません。

12.3.1 再配布とサポートされる STLport ライブラリ

エンドユーザーオブジェクトコードライセンスの条件に基づいて、作成した実行可能ファイルまたはライブラリとともに再配布可能なライブラリおよびオブジェクトファイルの一覧は、再配布の README を参照してください。この README の C++

セクションに、このリリースのコンパイラがサポートしている STLport.so のバージョンが記載されています。この Readme は、http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html にある、Oracle Solaris Studio ソフトウェアのこのリリースについての法的ページにあります。

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。

12.4 Apache stdcxx 標準ライブラリ

-library=stdcxx4 でコンパイルすることによって、デフォルトの libCstd の代わりに Apache stdcxx バージョン 4 C++ 標準ライブラリを Solaris で使用してください。このオプションにより、-mt オプションも暗黙的に設定されます。stdcxx ライブラリには、マルチスレッドモードが必要です。このオプションは、コンパイルのたびに、およびアプリケーション全体のリンクコマンドで一貫して使用する必要があります。-library=stdcxx4 を使用してコンパイルされたコードは、デフォルトの -library=Cstd または省略可能な -library=stlport4 を使用してコンパイルされたコードと同じプログラムでは使用できません。

◆ ◆ ◆ 第 13 章

13 従来の iostream ライブラリの使用

C++ も C と同様に組み込み型の入出力文はありません。その代わりに、出力機能はライブラリで提供されています。C++ コンパイラでは iostream クラスに対して、従来型の実装と ISO 標準の実装を両方とも提供しています。

- 互換モード (-compat[=4]) では、従来型の iostream クラスは libC に含まれています。
- 標準モード (デフォルトのモード) では、従来型の io stream クラスは libiostream に含まれています。従来型の libiostream 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、 iostream 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)。**man -s 3CC4** *name*

13.1 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 iostream は、cerr を除いて完全にバッファ利用します。178 ページの「13.3.1 iostream を使用した出力」と 180 ページの「13.3.2 iostream を使用した入力」を参照してください。

13.2 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できます。それぞれのストリームは、次のどれかを入力先または出力先とします。

- 標準入力
- 標準出力
- 標準エラー
- ファイル
- 文字型配列

ストリームは、入力のみまたは出力のみと制限して使用することも、入出力両方に使用することもできます。iostream ライブラリでは、次の 2 つの処理階層を使用してこのようなストリームを実現しています。

- 下層では、単なる文字ストリームであるシーケンスを実現します。シーケンスは、streambuf クラスか、その派生クラスで実現されています。
- 上層では、シーケンスに対してフォーマット操作を行います。フォーマット操作は istream と ostream の 2 つのクラスで実現されます。これらのクラスはメンバーに streambuf クラスから派生したオブジェクトを持っています。このほかに、入出力両方が実行されるストリームに対しては iostream クラスがあります。

標準入力、標準出力、標準エラーは、istream または ostream から派生した特殊なクラスオブジェクトで処理されます。

ifstream、ofstream、fstream の 3 つのクラスはそれぞれ istream、ostream、iostream から派生しており、ファイルへの入出力を処理します。

istrstream、ostrstream、strstream の 3 つのクラスはそれぞれ istream、ostream、および iostream から派生しており、文字型配列への入出力を処理します。

入力ストリームまたは出力ストリームをオープンする場合は、どれかの型のオブジェクトを生成し、そのストリームのメンバー streambuf をデバイスまたはファイルに関連付けます。通常、関連付けはストリームコンストラクタで行うので、ユーザーが直接 streambuf を操作することはありません。標準入力、標準出力、エラー出力に対しては、iostream ライブラリであらかじめストリームオブジェクトを定義してあるので、これらのストリームについてはユーザーが独自にオブジェクトを生成する必要はありません。

ストリームへのデータの挿入 (出力)、ストリームからのデータの抽出 (入力)、挿入または抽出したデータのフォーマット制御には、演算子または iostream のメンバー関数を使用します。

新たなデータ型 (ユーザー定義のクラス) を挿入したり抽出したりするときは一般に、挿入演算子と抽出演算子の多重定義をユーザーが行います。

13.3 従来の iostream ライブラリの使用

従来型の iostream ライブラリからルーチンを使用するには、必要なライブラリ部分のヘッダーファイルをインクルードする必要があります。 次の表で各ヘッダーファイルについて説明します。

表 13–1 iostream ルーチンのヘッダーファイル

ヘッダーファイル	内容の説明
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。
iomanip.h	マニピュレータ値の宣言。マニピュレータ値とはiostream に挿入または iostream から抽出する値で、特別の効果を引き起こします。この中で iostream.h をインクルードします。
stdiostream.h	(廃止) stdio FILE の使用に固有の iostream と streambuf の宣言。この中で iostream.h. をインクルードします。
stream.h	(旧形式) この中で iostream.h、fstream.h、iomanip.h、stdiostream.h をインクルードします。C++ Version 1.2 の旧形式ストリームと互換性を保つための宣言。

これらのヘッダーファイルすべてをプログラムにインクルードする必要はありません。自分のプログラムで必要な宣言の入ったものだけをインクルードします。互換モード (-compat[=4]) では、従来型の iostream ライブラリは libC の一部であり、CC ドライバによって自動的にリンクされます。標準モード (デフォルトのモード) では、従来型の libiostream ライブラリは iostream に含まれています。

13.3.1 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 ライブラリは、ユーザー定義の型をサポートしていません。独自の方法で出力しようとする型を定義する場合は、それらを正しく処理する挿入子を定義する (つまり、<<operator を多重定義する) 必要があります。

<< 演算子は反復使用できます。2 つの値を cout に挿入するには、次の例のような文を使用できます。

```
cout << someValue << anotherValue;
```

前述の例では、2 つの値の間に空白が入りません。空白を入れる場合は、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。ほかの演算子と同様に、括弧を使用して実行順序を指定できます。 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、あとの 2 つは異なります。

```
cout << a+b;              // + has higher precedence than <<
cout << (a+b);
cout << (a&y);            // << has precedence higher than &
cout << a&y;            // probably an error: (cout << a) & y
```

13.3.1.1 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:
    // (functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

この例では、string クラスのデータ部が private であるため、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。

```
ostream& operator<< (ostream& ostr, const string& output)
{    return ostr << output.data;}
```

前述の定義は、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できます。

13.3.1.2 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、iostream の状態も変わりません。

一般的なエラー処理方法は、メインのどこかで定期的に出力ストリームの状態を検査する方法です。そこで、エラーが起こっていることがわかれば、何らかの処理を行います。この章では、文字列を出力してプログラムを中止させる関数 error をユーザーが定義しているものとして説明します。error は事前定義された関数ではありません。error 関数の例は、183 ページの「13.3.9 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。iostream がエラー状態の場合はゼロ以外の値を返します。次に例を示します。

```
if (!cout) error("output error");
```

エラーを調べるにはもう 1 つの方法があります。ios クラスでは、operator void *() が定義されており、エラーが起こった場合は NULL ポインタを返します。したがって、次の文でエラーを検査できます。

```
if (cout << x) return; // return if successful
```

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; // return if successful
```

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。

13.3.1.3 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。出力バッファーをフラッシュする場合、次のように特殊な値 flush を挿入するだけでフラッシュできます。次に例を示します。

```
cout << "This needs to get out immediately." << flush;
```

flush は、マニピュレータと呼ばれるタイプのオブジェクトの 1 つです。マニピュレータを iostream に挿入すると、その値が出力されるのではなく、何らかの効果が引き起こされます。マニピュレータは実際には関数で、ostream& または istream& を引数として受け取り、そのストリームに対する何らかの動作を実行したあとにその引数を返します。188 ページの「13.7 マニピュレータ」を参照してください。

13.3.1.4 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、前述の例で出力した値を正しく読み込むことができません。

13.3.2 iostream を使用した入力

iostream を使用した入力は、出力と同じです。入力には、抽出演算子 >> を使用します。挿入演算子と同様に繰り返し指定できます。次に例を示します。

```
cin >> a >> b;
```

この例では、標準入力から 2 つの値が取り出されます。ほか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 (a と b の型が異なれば、別の抽出子が使用されます) の型によって決まります。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。

13.3.3 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。

例 13–1 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 (前述の例では 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ある必要があります。

13.3.4 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;
```

前述の例で、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; // not initialized
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。

13.3.5 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。次に例を示します。

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

注 – ほか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ほか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 ==' ');
```

13.3.6 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));
```

13.3.7 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。次に例を示します。

```
if (cin.peek()!= c) return 0;
```

13.3.8 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、のちにオンに戻しています。

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
...
cin.setf(ios::skipws); // turn it on again
```

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;
```

13.3.9 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。! は、iostream のエラー状態を検査します。たとえば次のコーディング例では、英字を入力すると入力エラーが発生します。

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
     cerr << message << "\n";
     exit(1);
}
int main() {
     cout << "Enter some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
     return 0;
}
```

クラス ios には、エラー処理に使用できるメンバー関数があります。詳細はマニュアルページを参照してください。

13.3.10 iostream と stdio の併用

C++ プログラムでも stdio を使用できますが、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 stdio 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 から入力しないかぎりは cin からも読み込むことができます。

13.4 iostream の作成

定義済みの iostream 以外のストリームを読み込む、あるいは書き込む場合は、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。

13.4.1 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、ofstrea m、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、そのほか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 をインクルードします。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。
- 新しい 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、プログラムの別の場所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。
- 上記のように、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。192 ページの「13.10 streambuf」 と、sbufpub(3CC4) のマニュアルページを参照してください。

13.4.1.1 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 で構築されます。open_mode は、ios クラスの公開部であり、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);
```

13.4.1.2 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、のち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);
```

13.4.1.3 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();
}
```

13.4.1.4 ファイル記述子を使用したファイルのオープン

標準出力は整数 1 などのようにファイル記述子がわかっている場合は、次のようにファイルをオープンできます。

```
ofstream outfile;
outfile.attach(1);
```

fstream コンストラクタの 1 つにファイル名を指定してファイルをオープンしたり、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。

13.4.1.5 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 の親クラスであるため、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。
- enum 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();     //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x;               //write a value to file
aFile.seekp(original);    //return to original position
```

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。

```
aFile.seekp(-10, ios::end);
```

この例では、ファイルの最後から 10 バイトの位置に設定されます。

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

一方、次の例では現在位置から 10 バイト進められます。

注– テキストストリーム上での任意位置へのシーク動作はマシン依存になります。ただし、以前に保存した streampos の値にいつでも戻ることができます。

13.5 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。

13.6 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。

13.7 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があります。

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。

表 13–2 の説明どおり、定義済みマニピュレータは 13 個あります。この表で使用している文字の意味は次のとおりです。

- i は long 型です。
- n は int 型です。
- c は char 型です。
- istr は入力ストリームです。
- ostr は入力ストリームです。

表 13–2 iostream の定義済みマニピュレータ

	定義済みマニピュレータ	内容の説明
1	ostr << dec, istr >> dec	10 を基数とする整数変換を行います。
2	ostr << endl	改行文字 ('\n') を挿入し、ostream::flush() を呼び出します。
3	ostr << ends	null (0) 文字を挿入します。strstream 取引時に役に立ちます。
4	ostr << flush	ostream::flush() を呼び出します。
5	ostr << hex, istr >> hex	16 を基数とする整数変換を行います。
6	ostr << oct, istr >> oct	8 を基数とする整数変換を行います。
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)	変換の基数を n (0、8、10、16 のみ) に設定します。
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。

表 13–2 iostream の定義済みマニピュレータ (続き)

	定義済みマニピュレータ	内容の説明
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)`	詰め合わせる文字 (フィールドのパディング用) を c に設定します。
13	`ostr << setprecision(n), istr >> setprecision(n)`	浮動小数点の精度を n 桁に設定します。

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&` のいず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&` のいずれかと、そのほかもう 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。次に、それぞれのタイプのマニピュレータの例を示します。

13.7.1 引数なしのマニピュレータの使用法

引数なしのマニピュレータは、次の 3 つを実行する関数です。

- ストリームの参照引数を受け取ります。
- そのストリームに何らかの処理を行います。
- その引数を返します。

`iostream` では、このような関数 (へのポインタ) を使用するシフト演算子がすでに定義されていますので、関数を入出力演算子シーケンスの中に入れることができます。シフト演算子は、値の入出力を行う代わりに、その関数を呼び出します。`tab` を `ostream` に挿入する `tab` マニピュレータの例を示します。

```
ostream& tab(ostream& os) {
              return os <<'\t';
            }
...
cout << x << tab << y;
```

次のコードは、前述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、オフを設定すると仮定します。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;
}
```

13.7.2 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。

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
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。
- 適用子。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 は次のことを行います。

- 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);
   }
```

13.8 ストリーム: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。

13.9 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。

13.10 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 を経由して streambuf を使用するため、streambuf の詳細について意識する必要はありません。たとえば、効率を改善する必要がある場合や、エラー処理を回避する場合、iostream に書式を構築する場合は、streambuf を直接使用できます。

13.10.1 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 – 次の文字が格納される直前を指します。
- *get* ポインタ - 取得される次の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。

13.10.1.1 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。

13.10.2 streambuf の使用

ユーザーは streambuf オブジェクト自体を作成することはなく、streambuf クラスから派生したクラスのオブジェクトを作成します。その例として、filebuf と strstreambuf とがあります。この 2 つについてはそれぞれ filebuf(3CC4) および ssbuf(3) のマニュアルページを参照してください。より高度な使い方として、独自のクラスを streambuf から派生させて特殊デバイスのインタフェースを提供したり、基本的なバッファリング以外のバッファリングを行なったりすることができます。sbufpub(3CC4) と sbufprot (3CC4) のマニュアルページでは、それらの方法について説明しています。

ユーザー用の特殊な streambuf を作成するとき以外にも、前述の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する場合があります。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。

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

入力ファイルと出力ファイルは、前述の例と同じ方法でオープンします。各 iostream クラスにはメンバー関数 rdbuf があり、それに結合した streambuf オブジェクトへのポインタを返します。fstream の場合、streambuf オブジェクトは filebuf 型です。fromFile に結合したファイル全体が toFile に結合したファイルにコピー (挿入) されます。最後の行は次のように書くこともできます。

```
fromFile >> toFile.rdbuf();
```

前述の書き方では、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。

13.11 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 13–3 iostream に関するマニュアルページの概要

マニュアルページ	概要
filebuf	streambuf から派生し、ファイル処理のために特殊化された filebuf クラスの公開インタフェースを詳細に説明します。streambuf クラスから継承した機能の詳細については、sbufpub(3CC4) と sbufprot(3CC4) のマニュアルページを参照してください。filebuf クラスは、fstream クラスを通して使用します。
fstream	istream、ostream、iostream をファイル処理用に特殊化した ifstream、ofstream、fstream の各クラスの特殊化されたメンバー関数を詳細に説明します。
ios	iostream の基底クラスである ios クラスの各部を詳細に説明します。すべてのストリームに共通の状態データについても説明します。
ios.intro	iostream を紹介し、概要を説明します。
istream	次の各項目を詳細に説明します。 ■ streambuf から取り出した文字の解釈をサポートする、クラス istream のメンバー関数 ■ 入力の書式設定 ■ ostream の一部として記述されている位置決め関数 ■ 一部の関連関数 ■ 関連マニピュレータ
manip	iostream ライブラリで定義されている入出力マニピュレータを説明します。
ostream	次の各項目を詳細に説明します。 ■ streambuf から取り出した文字の解釈をサポートする、クラス ostream のメンバー関数 ■ 出力の書式設定 ■ ostream の一部として記述されている位置決め関数 ■ 一部の関連関数 ■ 関連マニピュレータ
sbufprot	streambuf(3CC4) クラスから派生したクラスをコーディングするプログラマに必要なインタフェースを説明します。sbufprot(3CC4) のマニュアルページで説明されていない公開関数があるため、sbufpub(3CC4)のマニュアルページも参照してください。

表 13–3 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。streambuft 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	stdiobuf クラスに関する最小限の説明があります。このクラスは streambuf から派生したもので、stdio FILE の処理に特殊化されています。streambuf クラスから継承した機能の詳細は、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。

13.12 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 13–4 iostream の用語

iostream 用語	定義
バッファー	バッファーには、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ーで、もう 1 つは入出力一般に適用されるバッファーです。 iostream ライブラリに固有のバッファーは、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ーは、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、バッファーがいっぱいになるか、バッファー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ー」とは、前述で定義したように一般にいうバッファーがない streambuf を指します。この章では、streambuf を指す「バッファー」という用語を使用することを避けています。ただし、マニュアルページや C++ の他のドキュメントでは、 streambuf を意味する「バッファー」という用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クラスを指します。
挿入	iostream に出力データを送り込む操作を挿入といいます。
iostream	一般には、入力ストリームまたは出力ストリームです。
iostream ライブラリ	ファイル iostream.h、fstream.h、strstream.h、iomanip.h、ライブラリ stdiostream.h をインクルードすることにより使用できるライブラリです。iostream はオブジェクト指向のライブラリであるため、ユーザーが必要に応じて拡張できます。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ーで、put ポインタまたは get ポインタ、あるい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、特定のクラスを意味します。そのほか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。

表 13–4 iostream の用語 (続き)

iostream 用語	定義
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、特定のクラスを意味します。

◆ ◆ ◆ 第 14 章

14 複素数演算ライブラリの使用

複素数には「実部」と「虚部」があります。次に例を示します。

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

縮退の場合、`0 + 3i` のように完全に虚部だけのものは通常 `3i` と書き込み、`5 + 0i` のように完全に実部だけのものは通常 `5` と書き込みます。データ型 `complex` を使用すると複素数を表現できます。

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。

14.1 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次のものが含まれています。

- 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ほかに、 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。

14.1.1 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。

14.2 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。

14.2.1 complex クラスのコンストラクタ

complex には 2 つのコンストラクタがあります。それぞれの定義を次に示します。

```
complex::complex()            {re=0.0; im=0.0;}
complex::complex(double r, double i = 0.0) {re=r; im=i;}
```

複素数の変数を引数なしで宣言すると、最初のコンストラクタが使用され、実部も虚部もゼロで初期化されます。次の例では、実部も虚部もゼロの複素数の変数が生成されます。

```
complex aComp;
```

1 つまたは 2 つのパラメータを指定できます。いずれの場合にも、2 番目のコンストラクタが使用されます。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。次に例を示します。

```
complex aComp(4.533);
```

複素数の値は次のようになります。

```
4.533 + 0i
```

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。次に例を示します。

```
complex aComp(8.999, 2.333);
```

複素数の値は次のようになります。

```
8.999 + 2.333i
```

また、複素数ライブラリが提供する polar 関数を使用して複素数を生成することもできます。202 ページの「14.3 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。

14.2.2 算術演算子

複素数ライブラリでは、すべての基本算術演算子が定義されています。特に、次の 5 つの演算子は通常の型の演算と同様に使用することができ、優先順序も同じです。

```
+ - / * =
```

演算子 - は、通常の型の場合と同様に 2 項演算子としても単項演算子としても使用できます。

このほか、次の演算子の使用方法も通常の型で使用する演算子と同様です。

- 加算代入演算子 (+=)
- 減算代入演算子 (-=)
- 乗算代入演算子 (*=)
- 除算代入演算子 (/=)

ただし、この 4 つの演算子については、式の中で使用可能な値は生成されません。したがって、次のコードは機能しません。

```
complex a, b;
...
if ((a+=2)==0) {...}; // illegal
b = a *= b; // illegal
```

また、等しいか否かを判定する 2 つの演算子 (==、 !=) は、通常の型で使用する演算子と同様に使用することができます。

算術式で実数と複素数が混在しているときは、C++ では複素数のための演算子関数が使用され、実数は複素数に変換されます。

14.3 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、関数はオーバーフローが実際に起こるのを避け、代わりにメッセージで `complex_error` を呼び出します。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 14–1 複素数ライブラリの関数

複素数ライブラリ関数	内容の説明
`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 14–2 複素数の数学関数と三角関数

複素数ライブラリ関数	内容の説明
`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。

表 14–2 複素数の数学関数と三角関数 (続き)

複素数ライブラリ関数	内容の説明
`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)`
	引数を 2 つ持ちます。pow(*b*, *exp*).*b* を *exp* 乗します。

`complex sin(const complex)`	引数の正弦を返します。
`complex sinh(const complex)`	引数の双曲線正弦を返します。
`complex sqrt(const complex)`	引数の平方根を返します。
`complex tan(const complex)`	引数の正接を返します。
`complex tanh(const complex)`	引数の双曲線正接を返します。

14.4 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、cplxerr(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。

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。

14.5 入出力

d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&); //inserter
istream& operator>>(istream&, complex&); //extractor
```

抽出子と挿入子の基本的な説明については、176 ページの「13.2 iostream 操作の基本構造」と178 ページの「13.3.1 iostream を使用した出力」を参照してください。

入力の場合、複素数の抽出子 >> は、(括弧の中にあり、コンマで区切られた) 一組の値を入力ストリームから抽出し、複素数オブジェクトに読み込みます。最初の値が実部の値、2 番目の値が虚部の値となります。たとえば、次のような宣言と入力文がある場合、

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

(3.45, 5) と入力すると、複素数 x の値は 3.45 + 5.0i となります。抽出子の場合はこの反対になります。complex x(3.45, 5), cout<<x の場合は、(3.45, 5) と印刷されます。

入力データは、通常括弧の中でコンマで区切られた一組の値で、スペースは入れても入れなくてもかまいません。値を 1 つだけ入力したとき (括弧とスペースは入力してもしなくても同じ) は、抽出子は虚部をゼロとします。シンボル i を入力してはいけません。

挿入子は、複素数の実部と虚部をコンマで区切り、全体を括弧で囲んで挿入します。シンボル i は含まれません。2 つの値は double 型として扱われます。

14.6 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。算術型は自動的に complex 型に変換されます。算術演算子とほとんどの数学関数の complex バージョンがあります。次に例を示します。

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

b+i という式は混合算術演算です。整数 i は、コンストラクタ complex::complex(double,double=0) によって、complex 型に変換されます。このとき、まず整数から double 型に変換されます。y、double で割られる計算結果です。つまり、y は complex 型へ変換され、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。次も同様です。

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; // error: operator < cannot be applied to type complex
a >= b; // error: operator >= cannot be applied to type complex
```

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);      // error: no match for f(complex)
```

14.7 効率

クラス complex は効率も考慮して設計されています。

非常に簡単な関数が inline で宣言されており、関数呼び出しのオーバーヘッドをなくしています。

効率に差があるものは、関数が多重定義されています。たとえば、pow 関数には引数が complex 型のもののほかに、引数が double 型と int 型のものがあります。その方が double 型と int 型の計算がはるかに簡単になるからです。

complex.h をインクルードすると、C の標準数学ライブラリヘッダー math.h も自動的にインクルードされます。C++ の多重定義の規則により、次のようにもっとも効率の良い式の評価が行われます。

```
double x;
complex x = sqrt(x);
```

この例では、標準数学関数 sqrt(double) が呼び出され、その計算結果が complex 型に変換されます。最初に complex 型に変換され、sqrt(complex) が呼び出されるのではありません。これは、多重定義の解決規則から決まる方法で、もっとも効率の良い方法です。

14.8 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 14–3 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 15 章

15 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。

15.1 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。共有する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、プログラマはほか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、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、一般的に接尾辞として .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
- アプリケーションをリンクし直さずに、実行時にライブラリを置き換えることができる(これは、プログラムの再リンクおよび再配布の必要なしに、Solaris オペレーティングシステムの多くの改良をプログラムで利用できる主要機構である)
- `dlopen()` 関数呼び出しl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
- 共有ライブラリを異なる場所に移動すると、システムがライブラリを見つけられず、プログラムを実行できないことがある(この問題は、環境変数 `LD_LIBRARY_PATH` で解決できる)。

15.2 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。次に例を示します。

```
% CC -c foo.cc # Compile main file, templates objects are created.
% CC -xar -o foo.a foo.o # Gather all objects into a library.
```

`-xar` フラグによって、`CC` が静的(アーカイブ)ライブラリを作成します。`-o` 命令は、新しく作成するライブラリの名前を指定するために必要です。コンパイラは、コマンド行のオブジェクトファイルを調べ、これらのオブジェクトファイルと、テンプレートリポジトリで認識されているオブジェクトファイルとを相互参照します。そして、ユーザーのオブジェクトファイルに必要なテンプレートを(本体のオブジェクトファイルとともに)アーカイブに追加します。

注 – 既存のアーカイブのみを作成または更新するには、-xar フラグを使用します。このフラグをアーカイブの保守に使用しないでください。-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。

15.3 動的 (共有) ライブラリの構築

動的 (共有) ライブラリの構築方法は、コマンド行に -xar の代わりに -G を指定することを除けば、静的 (アーカイブ) ライブラリの場合と同じです。

ld は直接使用しないでください。静的ライブラリの場合と同じように、CC コマンドを使用すると、必要なすべてのテンプレートインスタンスがテンプレートリポジトリからライブラリに組み込まれます (テンプレートを使用している場合)。アプリケーションにリンクされている動的ライブラリでは、すべての静的コンストラクタは main() が実行される前に呼び出され、すべての静的デストラクタは main() が終了したあとに呼び出されます。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 libfoo.so 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ています。-o オプションは、ライブラリのファイル名を指定しています。-h オプションは、共有ライブラリの内部名を指定しています。-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。

CC -G コマンドは -l オプションをリンカー ld に渡しません。初期化順序が必ず正しくなるようにするには、共有ライブラリに自らが必要とするほかの各共有ライブラリとの明示的な依存関係を設定する必要があります。依存関係を作成するには、該当するライブラリごとに -l オプションを使用します。標準的な C++ 共有ライブラリは、次の一群のオプションのうち 1 つを使用します。

```
-lCstd -lCrun -lc
-library=stlport4 -lCrun -lc
```

必要とされるすべての依存関係をリストしたことを確認するには、-zdefs オプションを指定してライブラリを構築します。不明のシンボル定義ごとに、リンカーはエラーメッセージを生成します。不明の定義を指定するには、それらのライブラリに -l オプションを追加します。

不要な依存関係を含んでいるかどうか検索するには、次のコマンドを使用します。

```
ldd -u -r mylib.so
ldd -U -r mylib.so
```

その後、不要な依存関係を除外し、mylib.so を再構築できます。

15.4 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。

15.5 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ー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。

15.6 公開ライブラリの構築

ほか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` は使用しないようにします。

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。

15.7 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 (アプリケーションプログラミングインタフェース) 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の構築時に -lC を CC コマンド行に追加します。
- C インタフェースライブラリを標準モード (デフォルト) で構築している場合は、C インタフェースライブラリの構築時に -lCrun を cc ではなく CC コマンド行に追加します。

さらに、C++ 実行時ライブラリにもまったく依存しないようにするには、ライブラリソースに対して次のコーディング規則を適用する必要があります。

- どのような形式の new もしくは 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
- 実行時の型識別機構 (RunTime Type Information、RTTI) を使用しない

15.8 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) に依存していることを確認してください。

そのためには、共有ライブラリの構築時に、-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。次に例を示します。

```
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。

パート IV

付録

◆ ◆ ◆ 付 録 A

C++ コンパイラオプション

この付録では、C++ 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、特に記載がないかぎりすべてのプラットフォームに適用されます。SPARC システム版の Solaris に特有の機能は *SPARC*、x86 システム版の Solaris と Linux に特有の機能は *x86* として識別されます。Solaris OS のみに限定されている機能には *Solaris* というマークが付きます。Linux OS のみに限定されている機能には *Linux* というマークが付きます。Solaris OS に関する言及には、OpenSolaris OS も暗黙的に含まれることに注意してください。

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「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。

A.1 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。オプションがほかのオプションで置き換えられたり、ほかのオプションと同じである場合、詳細についてはほかのオプション説明を参照してください。

表 A–1 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あと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。

表 A–1 オプションの見出し (続き)

見出し	内容
デフォルト	オプションに一次または二次のデフォルト値がある場合は、それを示します。 一次のデフォルトとは、オプションが指定されなかったときに有効になるオプションの値です。たとえば、-compat を指定しないと、デフォルトは -compat=5 になります。 二次のデフォルトとは、オプションは指定されたが、値が指定されなかったときに有効になるオプションの値です。たとえば、値を指定せずに -compat を指定すると、デフォルトは -compat=4 になります。
拡張	オプションにマクロ展開がある場合は、ここに示します。
例	オプションの説明のために例が必要な場合は、ここに示します。
相互の関連性	ほか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。
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ほかのオプションや文書を示します。
「置き換え」、「同じ」	そのオプションが廃止され、ほか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、「-x0 と同じです」は、-x0 が望ましいオプションであることを示します。

A.2 オプションの一覧

次の節では、C++ コンパイラオプションのアルファベット順の一覧と、プラットフォームの制限を示しています。

A.2.1 -386

x86: `-xtarget=386` と同じです。このオプションは廃止され、下位互換のためだけに用意されています。

A.2.2 -486

x86: `-xtarget=486` と同じです。このオプションは廃止され、下位互換のためだけに用意されています。

A.2.3 -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)、静的 (共有でない) のいずれかから指定します。

`-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 の 64 ビットコンパイル環境では、多くのシステムライブラリは、動的ライブラリのみ使用できます。このため、コマンド行の最後に `-Bstatic` を使用しないでください。

A.2.3.1 値

binding には次のいずれかの値を指定します。

値	意味
`dynamic`	まず `lib`*lib*`.so` (共有) ファイルを検索するようにリンカーに指示します。これらのファイルが見つからないと、リンカーは `lib`*lib*`.a` (静的で共有されない) ファイルを検索します。ライブラリのリンク方式を共有にしたい場合は、このオプションを指定します。
`static`	-Bstatic オプションを指定すると、リンカーは `lib`*lib*`.a` (静的で、共有されない) ファイルだけを検索します。ライブラリのリンク形式を非共有にしたい場合は、このオプションを指定します。
`symbolic`	シンボルがほかですでに定義されている場合でも、可能であれば共有ライブラリ内でシンボル解決を実行します。 `ld`(1) のマニュアルページを参照してください。

-B と *binding* との間に空白があってはいけません。

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 を -Bdynamic にするべきです。

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには、libm.so および libc.so があります。libm.a と libc.a は提供していません。その結果、-Bstatic と -dn を使用すると 64 ビットの Solaris オペレーティングシステム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。ほか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)、149 ページの「11.5 標準ライブラリの静的リンク」、『リンカーとライブラリ』

A.2.4 -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。

A.2.4.1 例

CC -c x.cc と入力すると、x.o というオブジェクトファイルが生成されます。

CC -c x.cc -o y.o と入力すると、y.o というオブジェクトファイルが生成されます。

警告

コンパイラは、入力ファイル (.c、.i) に対するオブジェクトコードを作成する際に、.o ファイルを作業ディレクトリに作成します。リンク手順を省略すると、この .o ファイルは削除されません。

関連項目

-o *filename*、-xe

A.2.5 -cg{89|92}

(SPARC) 非推奨、使用しないでください。現在の Solaris オペレーティングシステムのソフトウェアは、SPARC V7 アーキテクチャーをサポートしません。このオプションでコンパイルすると、現在の SPARC プラットフォームでの実行速度が遅いコードが生成されます。-x0 を使用して、-xarch、-xchip、および -xcache のコンパイラのデフォルトを利用します。

A.2.6 –compat[={ 4|5|g}]

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 3 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。標準モードは、2003 年に更新された ANSI/ISO 1998 C++ 標準に従って構造体を受け入れます。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、そのほかの 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。–compat=g オプションにより、Linux の gcc/g++ コンパイラとの互換性が追加されます。

これらのモードは、-compat オプションで次に示す値を使用して指定します。

A.2.6.1 値

-compat オプションには次の値を指定できます。

値	意味
-compat=4	(互換モード) 言語とバイナリの互換性を 4.0.1、4.1、4.2 コンパイラに合わせます。__cplusplus プリプロセッサマクロを 1 に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 4 にそれぞれ設定します。
-compat=5	(標準モード) 言語とバイナリの互換性を ANSI/ISO 標準モード 5.0 コンパイラに合わせます。__cplusplus プリプロセッサマクロを 1997IIL に、__SUNPRO_CC_COMPAT プリプロセッサマクロを 5 にそれぞれ設定します。
-compat=g	(Linux のみ) g++ 言語拡張の認識を有効にし、Linux プラットフォームの g++ とバイナリ互換性があるコードを生成するように、コンパイラに指示します。__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 'G' にそれぞれ設定します。

-compat=g を使用すると、バイナリ互換性は個々の .o ファイルまたはアーカイブ (.a) ライブラリではなく、共有 (動的または .so) ライブラリにのみ拡張されます。

たとえば、g++ 共有ライブラリを C++ メインプログラムにリンクします。

```
% g++ -shared -o libfoo.so -fpic a.cc b.cc c.cc
% CC -compat=g main.cc -L. -lfoo
```

さらに、C++ 共有ライブラリを g++ メインプログラムにリンクします。

```
% CC -compat=g -G -o libfoo.so -Kpic a.cc b.cc c.cc
% g++ main.cc -L. -lfoo
```

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。

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。

- -Bsymbolic

- `-features=[no%]strictdestrorder`
- `-features=[no%]tmplife`
- `-library=[no%]iostream`
- `-library=[no%]Cstd`
- `-library=[no%]Crun`
- `-library=[no%]rwtools7_std`
- `-xarch=native64`、`-xarch=generic64`、`-xarch=v9`、`-xarch=v9a`、または `-xarch=v9b`

`-compat=5` では次のオプションの使用はサポートされません。

- `-Bsymbolic`
- `+e`
- `features=[no%]arraynew`
- `features=[no%]explicit`
- `features=[no%]namespace`
- `features=[no%]rtti`
- `library=[no%]complex`
- `library=[no%]libC`
- `-vdelx`

警告

共有ライブラリを構築するときは、`-Bsymbolic` を使用しないでください。

関連項目

『C++ 移行ガイド』

A.2.7 +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
- `–xO`*n* 最適化レベルが指定されずに `–g` オプションが選択されている

A.2.7.1 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;} // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() {return 0;} // may be inlined
};
```

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、`-xO4` または `-xO5` 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`

A.2.8 -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。

コンパイラの定義済みマクロのリストについては、`CC`(1) のマニュアルページを参照してください。

A.2.9 -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。

A.2.9.1 値

値	意味
`-dy`	リンカーで動的リンクを実行します。
`-dn`	リンカーで静的リンクを実行します。

デフォルト

`-d` オプションを指定しないと、`-dy` が使用されます。

相互の関連性

64 ビットの環境では、多くのシステムライブラリは共有の動的ライブラリとしてのみ利用できます。これらのシステムライブラリには、`libm.so` および `libc.so` があります。`libm.a` と `libc.a` は提供していません。その結果、`-Bstatic` と `-dn` を使用すると 64 ビットの Solaris オペレーティングシステム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。

警告

このオプションを動的ライブラリと組み合わせて使用すると、重大なエラーが発生します。ほとんどのシステムライブラリは、動的ライブラリでのみ使用できます。

関連項目

`ld`(1)、『リンカーとライブラリ』

A.2.10 `-dalign`

(SPARC) `-dalign` は、`-xmemalign=8s` を指定することと同じです。詳細は、326 ページの「A.2.151 `-xmemalign=`*ab*」を参照してください。

x86 プラットフォームでは、このオプションはメッセージを表示せずに無視されます。

A.2.10.1 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` でコンパイルします。コンパイルしない場合予期しない結果が生じることがあります。

A.2.11 -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。

A.2.12 -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。

ソースコードにテンプレートが含まれている場合に -E オプションの出力をコンパイルするには、-E オプションとともに -template=no%extdef オプションを使用する必要が生じることがあります。アプリケーションコードで「定義分離」テンプレートのソースコードモデルが使用されている場合、それでも -E オプションの出力がコンパイルされない可能性があります。詳細は、テンプレートの章を参照してください。

A.2.12.1 例

このオプションは、プリ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、次に示すプログラムでは、foo.cc は、224 ページの「A.2.12.1 例」に示す出力を生成します。

例 A–1 プリ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);
}
.
```

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);
```

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 (続き)

```
int main () {
int x;
x = power (2, 10);
}
```

警告

コードの中に「定義分離」モデルのテンプレートが含まれている場合は、このオプションの結果を C++ コンパイラの入力に使用できないことがあります。

関連項目

-P

A.2.13 +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。

A.2.13.1 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』

A.2.14 -erroff[= *t*]

このコマンドは、C++ コンパイラの警告メッセージを無効にします。エラーメッセージには影響しません。このオプションは、-errwarn でゼロ以外の終了状態を発生させるように指定されているかどうかにかかわらず、すべての警告メッセージに適用されます。

A.2.14.1 値

t には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。指定順序によって実行内容が異なります。たとえば、「%all,no%*tag*」と指定すると、*tag* 以外のすべての警告メッセージを抑制します。次の表は、-erroff の値を示しています。

表 A–2 -erroff の値

値	意味
tag	*tag* のリストに指定されているメッセージを抑制します。-errtags=yes オプションで、メッセージのタグを表示することができます。
no%*tag*	*tag* 以外のすべての警告メッセージの抑制を解除します。
%all	すべての警告メッセージを抑制します。
%none	すべてのメッセージの抑制を解除します (デフォルト)。

デフォルト

デフォルトは -erroff=%none です。-erroff と指定すると、-erroff=%all を指定した場合と同じ結果が得られます。

例

たとえば、-erroff=*tag* は、この tag が指定する警告メッセージを抑止します。一方、-erroff=%all,no%*tag* は、*tag* が識別するメッセージ以外の警告メッセージをすべて抑止します。

警告メッセージのタグを表示するには、-errtags=yes オプションを使用します。

警告

-erroff オプションで無効にできるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

関連項目

-errtags、-errwarn

A.2.15 -errtags[= *a*]

C++ コンパイラのフロントエンドで出力される警告メッセージのうち、-erroff オプションで無効にできる、または -errwarn オプションで重大な警告に変換できるメッセージのメッセージタグを表示します。

A.2.15.1 値とデフォルト

a には yes または no のいずれかを指定します。デフォルトは -errtags=no です。-errtags だけを指定すると、-errtags=yes を指定するのと同じことになります。

警告

C++ コンパイラのドライバおよび C のコンパイルシステムのほかのコンポーネントから出力されるメッセージにはエラータグが含まれないため、-erroff で無効にしたり、-errwarn で重大なエラーに変換したりすることはできません。

関連項目

-erroff、-errwarn

A.2.16 -errwarn[= *t*]

指定した警告メッセージが生成された場合に、重大なエラーを出力して C++ コンパイラを終了する場合は、-errwarn を使用します。

A.2.16.1 値

t には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。このとき、順序が重要になります。たとえば、%all,no%*tag* と指定すると、*tag* 以外のすべての警告メッセージが生成された場合に、重大なエラーを出力して cc を終了します。

-errwarn の値を次の表に示します。

表 A–3 -errwarn の値

値	意味
tag	*tag* に指定されたメッセージが警告メッセージとして発行されると、cc は致命的エラーステータスを返して終了します。*tag* に指定されたメッセージが発行されない場合は無効です。

表 A–3 -errwarn の値　(続き)

値	意味
no%*tag*	*tag* に指定されたメッセージが警告メッセージとしてのみ発行された場合に、CC が致命的なエラーステータスを返して終了しないようにします。*tag* に指定されたメッセージが発行されない場合は無効です。このオプションは、*tag* または %all を使用して以前に指定したメッセージが警告メッセージとして発行されても cc が致命的エラーステータスで終了しないようにする場合に使用してください。
%all	警告メッセージが 1 つでも発行されると CC は致命的ステータスを返して終了します。%all に続いて no%*tag* を使用して、特定の警告メッセージを対象から除外することもできます。
%none	どの警告メッセージが発行されても CC が致命的エラーステータスを返して終了することがないようにします。

デフォルト

デフォルトは -errwarn=%none です。-errwarn だけを指定した場合、-errwarn=%all を指定したことと同じになります。

警告

-errwarn オプションを使用して、障害状態で C++ コンパイラを終了するように指定できるのは、C++ コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

C++ コンパイラで生成される警告メッセージは、コンパイラのエラーチェックの改善や機能追加に応じて、リリースごとに変更されます。-errwarn=%all を指定してエラーなしでコンパイルされるコードでも、コンパイラの次期リリースではエラーを出力してコンパイルされる可能性があります。

関連項目

-erroff、-errtags、-xwe

A.2.17 -fast

このオプションは、 実行ファイルの実行時のパフォーマンスのチューニングで効果的に使用することができるマクロです。-fast は、コンパイラのリリースによって変更される可能性があるマクロで、ターゲットのプラットフォーム固有のオプションに展開されます。-dryrun オプションまたは -xdryrun を使用して -fast の展開を調べ、-fast の該当するオプションを使用して実行可能ファイルのチューニングを行なってください。

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。

A.2.17.1 拡張

このオプションは、次のコンパイラオプションを組み合わせて、多くのアプリケーションのパフォーマンスをほぼ最大にします。

表 A–4 -fast の拡張

オプション	SPARC	x86
-fns	X	X
-fsimple=2	X	X
-nofstore	-	X
-xarch	X	X
-xbuiltin=%all	X	X
-xcache	X	X
-xchip	X	X
-xlibmil	X	X
-xlibmopt	X	X
-xmemalign	X	-
-xO5	X	X
-xregs=frameptr	-	X
-xtarget=native	X	X

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、指定されたほか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ーは -xarch に指定された SPARC-V9 ではなく、32 ビットアーキテクチャー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、そのあと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。

x86 では、–fast オプションに –xregs=frameptr が含まれます。特に C、Fortran、および C++ の混合ソースコードをコンパイルする場合は、その詳細について、このオプションの説明を参照してください。

例

次のコンパイラコマンドでは、最適化レベルは -x03 になります。

```
example% CC –fast –xO3
```

次のコンパイラコマンドでは、最適化レベルは -xO5 になります。

```
example% CC -xO3 –fast
```

警告

別々の手順でコンパイルしてリンクする場合は、-fast オプションをコンパイルコマンドとリンクコマンドの両方に表示する必要があります。

-fast オプションでコンパイルしたオブジェクトバイナリは移植できません。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。

-fast の展開には、-D_MATHERR_ERRNO_DONTCARE が含まれます。

-fast を使用すると、コンパイラは errno 変数を設定しない同等の最適化コードを使用して呼び出しを浮動小数点関数に自由に置き換えることができます。さらに、-fast はマクロ __MATHERR_ERRNO_DONTCARE も定義します。このマクロを使用すると、コンパイラは errno の妥当性の確認を無視できます。この結果、浮動小数点関数の呼び出しのあとに errno の値に依存するユーザーコードにより、一貫しない結果が生成される可能性があります。

この問題を解決する 1 つの方法は、-fast を使用してそのようなコードをコンパイルしないことです。ただし、-fast の最適化が必要で、コードが浮動小数点ライブラリの呼び出しのあとに正しく設定される errno の値に依存している場合は、次のオプションを使用してコンパイルしてください。

```
-xbuiltin=none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

これを、コマンド行で -fast のあとに使用することで、コンパイラはそのようなライブラリ呼び出しを最適化しなくなり、errno が確実に正しく処理されるようになります。

任意のプラットフォームで –fast の展開を表示するには、 **CC –dryrun –fast** コマンドを実行します。

```
>CC -dryrun -fast
###     command line files and options (expanded):
### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 \
-xchip=ultra3i -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none \
-xlibmil -xlibmopt -xbuiltin=%all -D__MATHERR_ERRNO_DONTCARE
```

関連項目

-fns、 -fsimple、 -ftrap=%none、 -xlibmil、 -nofstore、 -xO5、 -xlibmopt、 -xtarget=native

A.2.18 -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。

A.2.18.1 値

互換モード (-compat[=4]) と標準モード (デフォルトモード) の両方で、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 -features オプション

値	意味
%all	指定されているモード (互換モードか標準モード) 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。

表 A–5 互換モードと標準モードでの -features オプション (続き)

値	意味
[no%]bool	ブール型とリテラルを許可します [しません]。有効にした場合、マクロ _BOOL=1 が定義されます。有効にしないと、マクロは定義されません。デフォルトは互換モードで no%bool、標準モードで bool です。
[no%]conststrings	リテラル文字列を読み取り専用メモリーに入れます [入れません]。デフォルトは互換モードで no%conststrings、標準モードで conststrings です。
[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。108 ページの「8.3 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。この機能はまだ実装されていませんが、export キーワードは認識されます。
[no%]extensions	ほかの 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]。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 文に対して標準準拠の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。
[no%]nestedacess	(標準モードのみ) ネストしたクラスが、包含するクラスの private メンバーにアクセスできるようにします [しません]。デフォルト: -features=nestedaccess
[no%]rvalueref	右辺値または一時値への const 以外の参照のバインドを許可します [しません]。デフォルト: -features=no%rvalueref デフォルトでは、C++ コンパイラは const 以外の参照を一時値または右辺値にバインドできないという規則を適用します。この規則を上書きするには、-features=rvalueref オプションを使用します。

表 A–5 互換モードと標準モードでの -features オプション (続き)

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]。-features=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かぎ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]。-features=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。-features=transitions を標準モードで使用すると、これらの言語構造に関してエラーメッセージではなく警告が出されます。-features=transitions を互換モード (-compat[=4]) で使用すると、コンパイラでは +w または +w2 が指定された場合にかぎ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

値	意味
[no%]strictdestrorder	静的記憶領域にあるオブジェクトを破棄する順序に関する、C++ 標準の必要条件に従います [従いません]。デフォルトは strictdestrorder です。
[no%]tmplrefstatic	関数テンプレートからの依存静的関数または静的関数テンプレートの参照を許可します [許可しません]。デフォルトは標準準拠の no%tmplrefstatic 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 compat=4 モードで no%tmplife、標準モードで 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

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。
[no%]rtti	実行時の型識別 (RTTI) を許可します [しません]。dynamic_cast<> および typeid 演算子を使用する場合は、RTTI を有効にする必要があります。-compat=4 mode の場合、デフォルトは no%rtti です。そうでない場合、デフォルトは -features=rtti で、オプション -features=no%rtti は使用できません。

注 – [no%]castop の設定は、C++ 4.2 コンパイラ用に作成されたメイクファイルとの互換性を維持するために使用できますが、それ以降のバージョンの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、互換モードのデフォルト (-compat[=4]) が使用されます。

```
-features=%none,anachronisms,except,split_init,transitions
```

デフォルトである「標準モード」では、

```
-features=%all,no%altspell,no%bool,no%conststrings,no%extensions,no%iddollar,\
              no%rvalueref,no%tmplrefstatic
```

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。

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ーと互換性がありません。

- `no%bool`
- `no%except`
- `no%mutable`
- `no%explicit`

互換モード (`-compat[=4]`) では、`+w` オプションまたは `+w2` オプションを指定しないかぎり、`-features=transitions` オプションは無効です。

警告

`-features=%all` や `-features=%none` を使用するときは注意してください。機能群がコンパイラおよびパッチのリリースのたびに変わる可能性があります。その結果、予期しない動作になる可能性があります。

`-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。

互換モード (`-compt=4`) の場合、デフォルトではコンパイラは `-features=split_init` と見なします。`-features=%none` オプションを使用してほか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

表 3–17 および『C++ 移行ガイド』

A.2.19 `-filt[=`*filter*`[,`*filter*`...]]`

コンパイラによってリンカーとコンパイラのエラーメッセージに通常適用されるフィルタリングを制御します。

A.2.19.1 値

filter は次の値のいずれかである必要があります。

表 A–8 -filt の値

値	意味
[no%]errors	C++ のリンカーエラーメッセージの説明を表示します [しません]。説明の抑止は、リンカーの診断を別のツールに直接提供している場合に便利です。
[no%]names	C++ で符号化されたリンカー名を復号化します [しません]。
[no%]returns	関数の戻り型を復号化します [しません]。この種の復号化を抑止すると、より迅速に関数名が識別しやすくなりますが、共有の不変式の戻り値の場合、一部の関数は戻り型でのみ異なることに注意してください。
[no%]stdlib	リンカーとコンパイラの両方のエラーメッセージに出力される標準ライブラリからの名前を簡略化します。この結果、標準ライブラリテンプレート型の名前を容易に認識できるようになります。
%all	-filt=errors,names,returns,stdlib に相当します。これはデフォルトの動作です。
%none	-filt=no%errors,no%names,no%returns,no%stdlib に相当します。

デフォルト

-filt オプションを指定しないで、または値を入れないで -filt を指定すると、コンパイラでは -filt=%all が使用されます。

例

次の例では、このコードを -filt オプションでコンパイルしたときの影響を示します。

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // no definition provided
};

int main()
{
    type t;
}
```

-filt オプションを指定しないでコードをコンパイルすると、コンパイラでは -filt=errors,names,returns,stdlib が使用され、標準出力が表示されます。

```
example% CC filt_demo.cc
Undefined             first referenced
 symbol                  in file
type::~type()         filt_demo.o
```

```
type::__vtbl          filt_demo.o
[Hint: try checking whether the first non-inlined, /
non-pure virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコマンドでは、C++ で符号化されたリンカー名の復号化が抑止され、C++ のリンカーエラーの説明が抑止されます。

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

次のコードについて考えてみましょう。

```
#include <string>
#include <list>
int main()
{
    std::list<int> l;
    std::string s(l); // error here
}
```

次は、`-filt=no%stdlib` を指定したときの出力です。

```
Error: Cannot use std::list<int, std::allocator<int>> to initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

次は、`-filt=stdlib` を指定したときの出力です。

```
Error: Cannot use std::list<int> to initialize std::string .
```

相互の関連性

`no%names` を使用しても `returns` や `no%returns` に影響はありません。つまり、次のオプションは同じ効果を持ちます。

- `-filt=no%names`
- `-filt=no%names,no%returns`
- `-filt=no%names,returns`

A.2.20 -flags

`-xhelp=flags` と同じです。

A.2.21 -fma[={none| fused}]

(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。-fma=none を指定すると、これらの命令の生成を無効にします。-fma=fused を指定すると、コンパイラは浮動小数点の積和演算 (FMA) 命令を使用して、コードのパフォーマンスを改善する機会を検出しようとします。

デフォルトは -fma=none です。

コンパイラが積和演算 (FMA) 命令を生成するための最小要件は、-xarch=sparcfmaf と、最適化レベルが -xO2 以上であることです。積和演算 (FMA) 命令をサポートしていないプラットフォームでプログラムが実行されないようにするため、コンパイラは積和演算 (FMA) 命令を生成する場合、バイナリプログラムにマーク付けをします。

積和演算 (FMA) により、積と和 (乗算と加算) の間で中間の丸め手順が排除されます。その結果、-fma=fused を指定してコンパイルしたプログラムは、精度は減少ではなく増加する傾向にありますが、異なる結果になることがあります。

A.2.22 -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な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。

A.2.22.1 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

x86: -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』

A.2.23 -fns[={yes| no}]

- SPARC: SPARC 非標準浮動小数点モードを有効または無効にします。

-fns=yes (または -fns) を指定すると、プログラムが実行を開始するときに、非標準浮動小数点モードが有効になります。

このオプションを使うと、-fns を含むほかのマクロオプション (-fast など) のあと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。

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。

- (*x86*) SSE flush-to-zero モードを選択します。利用可能な場合には、denormals-are-zero モードが選択されます。

  このオプションは、非正規数の結果をゼロにフラッシュします。また利用可能な場合には、非正規数オペランドもゼロとして扱われます。

  このオプションは、SSE や SSE2 命令セットを利用しない従来の x86 浮動小数点演算には影響しません。

A.2.23.1 値

-fns オプションには次の値を指定できます。

表 A–9 -fns の値

値	意味
yes	非標準浮動小数点モードを選択します。
no	標準浮動小数点モードを選択します。

デフォルト

-fns を指定しないと、非標準浮動小数点モードは自動的には有効にされません。標準の IEEE 754 浮動小数点計算が行われます。つまり、アンダーフローは段階的です。

-fns だけを指定すると、-fns=yes が想定されます。

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、そのあとに続く -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` オプション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。コンパイルしない場合、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。x86 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`) オプションを使用したときに、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)

A.2.24 `-fprecision=`*p*

x86: 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。

A.2.24.1 値

p は次のいずれかでなければいけません。

表 A–10 `-fprecision` の値

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 オプションを指定しないと、丸め精度モードは extended になります。

警告

このオプションは、x86 システムでメインプログラムのコンパイル時に使用する場合にのみ有効で、64 ビット (-m64) または SSE2 対応 (-xarch=sse2) プロセッサでコンパイルする場合は無視されます。SPARC システムでも無視されます。

A.2.25 -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。

A.2.25.1 値

r には次の値のいずれかを指定します。

表 A–11 -fround の値

値	意味
nearest	もっとも近い数値に丸め、中間値の場合は偶数にします。
tozero	ゼロに丸めます。
negative	負の無限大に丸めます。
positive	正の無限大に丸めます。

デフォルト

`-fround` オプションを指定しないと、丸めモードは `-fround=nearest` になります。

警告

1 つのルーチンを `-fround=`*r* でコンパイルした場合は、そのプログラムのすべてのルーチンを同じ `-fround=`*r* オプションでコンパイルする必要があります。コンパイルしない場合、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。

A.2.26 –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。

A.2.26.1 値

n を指定する場合、0、1、2 のいずれかにしなければいけません。

表 A–12 `-fsimple` の値

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。生成されるコードは IEEE 754 に厳密には準拠していませんが、大半のプログラムの数値結果は変わりありません。 `-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。

表 A–12 -fsimple の値 (続き)

値	意味
2	-fsimple=1 のすべての機能に加えて、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 z=1/y 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、コンパイラーは -fsimple=0 を使用します。

-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

最適化が精度に与える影響の詳細は、『*Techniques for Optimizing Applications: High Performance Computing*』(Rajat Garg、Ilya Sharapov 共著) をお読みください。

A.2.27 -fstore

x86:

浮動小数点式の精度を強制的に使用します。

このオプションを指定すると、コンパイラは、次の場合に浮動小数点の式や関数の値を代入式の左辺の型に変換します。つまり、その値はレジスタにそのままの型で残りません。

- 式や関数を変数に代入する。
- 式をそれより短い浮動小数点型にキャストする。

このオプションを解除するには、オプション -nofstore を使用してください。

A.2.27.1 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`

A.2.28 `-ftrap=`*t*`[,`*t*`...]`

起動時の IEEE トラップモードを設定します。ただし、`SIGFPE` ハンドラは組み込まれません。トラップの設定と SIGFPE ハンドラの組み込みを同時に行うには、`ieee_handler`(3M) か `fex_set_handling`(3M) 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。

A.2.28.1 値

t には次の値のいずれかを指定できます。

表 A–13 `-ftrap` の値

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`	前述のすべてをトラップします。
`%none`	前述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。`[no%]` 形式のオプション自体は、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。

デフォルト

`-ftrap` を指定しない場合、コンパイラは `-ftrap=%none` とみなします。

例

`-ftrap=%all,no%inexact` は、inexact を除くすべてのトラップが設定されます。

警告

1 つのルーチンを `-ftrap=`*t* オプションでコンパイルした場合は、そのプログラムのルーチンすべてを、`-ftrap=`*t* オプションを使用してコンパイルしてください。途中から異なるオプションを使用すると、予想に反した結果が生じることがあります。

`-ftrap=inexact` のトラップは慎重に使用してください。`-ftrap=inexact` では、浮動小数点の値が正確でないとトラップが発生します。たとえば、次の文ではこの条件が発生します。

```
x = 1.0 / 3.0;
```

このオプションは、メインプログラムをコンパイルするときにだけ有効です。このオプションを使用する際には注意してください。IEEE トラップを有効にするには `-ftrap=common` を使用してください。

関連項目

`ieee_handler`(3M) および `fex_set_handling`(3M) のマニュアルページ

A.2.29 -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 `-xcode=pic13` オプションでコンパイルされます。

テンプレートが含まれていて、`-instances=extern` オプションを使用してコンパイルされたファイルから共有ライブラリを構築すると、`.o` ファイルにより参照されているテンプレートインスタンスがすべてテンプレートキャッシュから自動的に含められます。

コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションと `-G` オプションを組み合わせて共有ライブラリを作成した場合は、生成された共有オブジェクトとのリンクでも、必ず同じオプションを指定してください。

300 ページの「A.2.118 -xcode=*a*」で推奨しているように、共有オブジェクトの作成では、`-xarch=v9` を付けてコンパイルしたすべてのオブジェクトファイルもまた、明示的な `-xcode` 値を付けてコンパイルする必要があります。

A.2.29.1 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、次のオプションがリンカーに渡されます。

- -dy
- -G
- -R

警告

共有ライブラリの構築には、ld -G ではなく、CC -G を使用してください。こうすると、CC ドライバによって C++ に必要ないくつかのオプションが ld に自動的に渡されます。

-G オプションを使用すると、コンパイラはデフォルトの -l オプションを ld に渡しません。共有ライブラリを別の共有ライブラリに依存させる場合は、必要な -l オプションをコマンド行に渡す必要があります。たとえば、共有ライブラリを libCrun に依存させる場合は、-lCrun をコマンド行に渡す必要があります。

関連項目

-dy、-Kpic、-xcode=pic13、-ztext、ld(1) のマニュアルページ、209 ページの「15.3 動的 (共有) ライブラリの構築」

A.2.30 -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。
- 「補助関数」を生成する。デバッガはこれを呼び出して、その一部の機能を実現する。
- 最適化レベルが指定されていない場合は、関数のインライン生成を無効にします。つまり、最適化レベルも指定されていない場合、このオプションを使用すると +d オプションが指定されていることになります。-O レベルまたは -xO レベルが指定された -g では、インラインは無効になりません。
- 特定のレベルの最適化を無効にする。

A.2.30.1 相互の関連性

このオプションと -x0*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、331 ページの「A.2.157 -x0*level*」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 -xO4 以上の場合、可能なかぎりのシンボリック情報と最高の最適化が得られます。最適化レベルを指定しないで –g を使用した場合、関数呼び出しのインライン化が無効になります (–g を使用して最適化レベルが指定されると、インラインが有効になります)。

このオプションを指定し、-O と -xO のどちらも指定していない場合は、+d +d オプションが自動的に指定されます。

注 – 以前のリリースでは、このオプションは、コンパイラのリンク専用の呼び出しにおいて、デフォルトで強制的にリンカー (ld) ではなく、インクリメンタルリンカー (ild) を使用するようにしていました。すなわち、-g が指定されたときのコンパイラは、そのデフォルトの動作として、コマンド行に -G またはソースファイルの指定がなくてもオブジェクトファイルのリンクで必ず、ld の代わりに ild を自動的に呼び出していました。現在、このようなことはありません。インクリメンタルリンカーは利用できなくなりました。

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分析機能は -g を必要としませんが、注釈付きのソースコード、一部の関数レベルの情報、およびコンパイラの注釈メッセージを確認するには、-g でコンパイルする必要があります。詳細は、analyzer(1) のマニュアルページと『プログラムのパフォーマンス解析』を参照してください。

-g オプションで生成される注釈メッセージは、プログラムのコンパイル時にコンパイラが実行した最適化と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、ほかの手順から -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影響します。-g (または -g0) でコンパイルされず、-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。通常、main 関数の入っているモジュールをデバッグするには、-g オプション (または -g0 オプション) でコンパイルする必要があります。

関連項目

+d、-g0、-xs、analyzer(1) マニュアルページ、er_src(1) マニュアルページ、および 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』

A.2.31 -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 が無効になり、インライン化された関数に dbx がステップインできなくなることを除けば、-g と同じです。

-xO3 以下の最適化レベルで -g0 を指定すると、ほとんど完全な最適化と可能なかぎりのシンボル情報を取得することができ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。

A.2.31.1 関連項目

+d、-g、『dbx コマンドによるデバッグ』

A.2.32 -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。

A.2.33 -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。動的共有ライブラリ

これはリンカー用のオプションで、ld に渡されます。通常、-h のあとに指定する name (名前) は、-o のあとに指定する名前と同じでなければいけません。-h と *name* の間には、空白文字を入れても入れなくてもかまいません。

コンパイルの時ローダーは、作成対象の共有動的ライブラリに、指定の名前を割り当てます。この名前は、ライブラリのイントリンシック名として、ライブラリファイルに記録されます。-h*name* (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。

-h オプションを指定せずに共有ライブラリを構築する場合は、実行時のローダーはライブラリのファイル名のみを検索します。ライブラリを、同じファイル名を持つほかのライブラリに置換することもできます。共有ライブラリにイントリンシック名がある場合は、ローダーはファイルを読み込むときにイントリンシック名を確認します。イントリンシック名が一致しない場合は、ローダーは置換ファイルを使用しません。

A.2.33.1 例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

A.2.34 -help

-xhelp=flags と同じです。

A.2.35 -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、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) を持つ #include ファイルを検索するためのディレクトリリストに、*pathname* (パス名) を追加します。

コンパイラは、引用符付きのインクルードファイル (#include "foo.h" の形式) 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++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊目的ファイルの include ディレクトリ内
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++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊目的ファイルの include ディレクトリ内
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、153 ページの「11.7.5 標準ヘッダーの実装」も参照してください。

A.2.35.1 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。151 ページの「11.7 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検索します。

`-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。

警告

コンパイラがインストールされている位置の `/usr/include`、 `/lib`、`/usr/lib` を検索ディレクトリに指定しないでください。

関連項目

`-I-`

A.2.36 -I-

インクルードファイルの検索規則を次のとおり変更します。

`#include "foo.h"` 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. `-I` オプションで指定されたディレクトリ内 (`-I-` の前後)

2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ。

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` のあとに指定した `-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、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ。

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の名前と一致する場合は、153 ページの「11.7.5 標準ヘッダーの実装」も参照してください。

A.2.36.1 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 を含むディレクトリから 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。代わりに、コマンド行に配置されている順番で、-I オプションで命名されたディレクトリを検索します。#include "c.h" 文を inc/a.h で処理するときは、コンパイラで ./c.h ヘッダーファイルが、inc/c.h ヘッダーファイルの代わりに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かぎ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。

コンパイラがインストールされている位置の /usr/include、 /lib、/usr/lib を検索ディレクトリに指定しないでください。

A.2.37 -i

リンカー ld は LD_LIBRARY_PATH の設定を無視します。

A.2.38 -include *filename;*

このオプションを指定すると、コンパイラは *filename* を、主要なソースファイルの 1 行目に記述されているかのように #include プリプロセッサ指令として処理します。ソースファイル t.c の考慮:

```
main()
{
   ...
}
```

t.c を **cc -include t.h t.c** コマンドを使用してコンパイルする場合は、ソースファイルに次の内容が含まれているかのようにコンパイルが進行します。

```
#include "t.h"
main()
{
   ...
}
```

コンパイラが *filename* を検索する最初のディレクトリは現在の作業ディレクトリであり、ファイルが明示的にインクルードされている場合のようにメインのソースファイルが存在するディレクトリになるわけではありません。たとえば、次のディレクトリ構造では、同じ名前を持つ 2 つのヘッダーファイルが異なる場所に存在しています。

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

作業ディレクトリが `foo/bar` であり、**`cc ../t.c -include t.h`** コマンドを使用してコンパイルする場合は、コンパイラによって `foo/bar` ディレクトリから取得された `t.h` がインクルードされますが、ソースファイル `t.c` 内で `#include` 指令を使用した場合の `foo/` ディレクトリとは異なります。

`-include` で指定されたファイルをコンパイラが現在の作業ディレクトリ内で見つけることができない場合は、コンパイラは通常のディレクトリパスでこのファイルを検索します。複数の `-include` オプションを指定する場合は、コマンド行で指定された順にファイルはインクルードされます。

A.2.39 -inline

`-xinline` と同じです。

A.2.40 -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。

A.2.40.1 値

a には次のいずれかを指定します。

表 A–14 -instances の値

値	意味
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の comdat セクション内に置き、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。 注: コンパイルとリンクを別々に行うとき、コンパイル処理で -instance=extern を指定した場合には、リンク処理でも -instance=extern を指定する必要があります。
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。
global	必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。
semiexplicit	明示的にインスタンス化されたインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。明示的なインスタンスにとって必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して大域リンケージを行います。必要なインスタンスがほかにあっても生成しません。
static	注: -instances=static は非推奨です。-instances=global が static の利点をすべて備えており、かつ欠点を備えていないので、-instances=static を使用する理由はなくなっています。このオプションは、このバージョンのコンパイラには存在しない、旧リリースのコンパイラにあった問題を克服するために用意されていました。 必要なすべてのインスタンスを現在のオブジェクトファイルに置き、それらに対して静的リンケージを行います。

デフォルト

-instances を指定しないと、-instances=global が想定されます。

関連項目

99 ページの「7.2.4 テンプレートインスタンスの配置とリンケージ」

A.2.41 -instlib=*filename*

このオプションを使用すると、ライブラリ (共有、静的) と現在のオブジェクトで重複するテンプレートインスタンスの生成が禁止されます。一般に、ライブラリを使用するプログラムが多数のインスタンスを共有する場合、-instlib=*filename* を指定して、コンパイル時間の短縮を試みることができます。

A.2.41.1 値

既存のテンプレートインスタンスが入っていることがわかっているライブラリを指定するには、*filename* 引数を使用します。ファイル名引数には、スラッシュ (/) 文字を含める必要があります。現在のディレクトリに関連するパスの場合には、ドットスラッシュ (./) を使用します。

デフォルト

`-instlib=`*filename* オプションにはデフォルト値はないので、値を指定する場合にのみ使用します。このオプションは複数回指定でき、指定内容は追加されていきます。

次に例を示します。

`libfoo.a` ライブラリと `libbar.so` ライブラリが、ソースファイル `a.cc` と共有する多数のテンプレートインスタンスをインスタンス化すると仮定します。`-instlib=`*filename* を追加してライブラリを指定すると、冗長性が回避されコンパイル時間を短縮できます。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

相互作用

`-g` を使ってコンパイルするとき、`-instlib=`*file* で指定したライブラリが `-g` でコンパイルされていない場合には、テンプレートインスタンスがデバッグ不能となります。この問題の対策としては、`-g` を指定するときに `-instlib=`*file* を使用しないようにします。

警告

`-instlib` によってライブラリを指定する場合には、そのライブラリとのリンクを行う必要があります。

関連項目

`-template`、`-instances`、`-pti`

A.2.42 -KPIC

SPARC: `-xcode=pic32` と同じです。

x86: `-Kpic` と同じです。

このオプションは、共有ライブラリを構築するためにソースファイルをコンパイルするときに使用します。大域データへの各参照は、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。

A.2.43 -Kpic

SPARC: `-xcode=pic13` と同じです。

x86: 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。

A.2.44 -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ます。

このオプションを `-verbose=diags` と一緒に使用すると、デバッグに便利です。

A.2.44.1 関連項目

`–v`、`–verbose`

A.2.45 -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。

A.2.45.1 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。

警告

コンパイラがインストールされている位置の `/usr/include`、 `/lib`、`/usr/lib` を検索ディレクトリに指定しないでください。

A.2.46 -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。

このオプションはファイル名のあとに使用してください。

A.2.46.1 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あとに -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。

関連項目

-L*dir*、-mt、134 ページの「10.4.8 アプリケーションの例」、『Tools.h++ クラスライブラリリファレンスマニュアル』

A.2.47 -libmieee

-xlibmieee と同じです。

A.2.48 -libmil

-xlibmil と同じです。

A.2.49 -library=/[,/...]

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。

A.2.49.1 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–15 互換モードに使用できる -library オプション

値	意味
[no%]f77	非推奨。-xlang=f77 を使用してください。
[no%]f90	非推奨。-xlang=f90 を使用してください。
[no%]f95	非推奨。-xlang=f95 を使用してください。
[no%]rwtools7	古い iostream Tools.h++ Version 7 を使用します [しません]。
[no%]rwtools7_dbg	デバッグ可能な Tools.h++ Version 7 を使用します [しません]。
[no%]complex	複素数の演算に libcomplex を使用します [しません]。
[no%]interval	非推奨。使用しないでください。-xia を使用してください。
[no%]libC	C++ サポートライブラリ libC を使用します [しません]。
[no%]gc	ガベージコレクション libgc を使用します [しません]。
[no%]sunperf	Sun Performance Library を使用します [しません]。
%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–16 標準モードに使用できる -library オプション

値	意味
[no%]f77	非推奨。-xlang=f77 を使用してください。
[no%]f90	非推奨。-xlang=f90 を使用してください。
[no%]f95	非推奨。-xlang=f95 を使用してください。
[no%]rwtools7	古い iostream Tools.h++ Version 7 を使用します [しません]。
[no%]rwtools7_dbg	デバッグ可能な Tools.h++ Version 7 を使用します [しません]。
[no%]rwtools7_std	標準 iostream Tools.h++ Version 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++ Version 7 を使用します [しません]。

表 A–16 標準モードに使用できる -library オプション (続き)

値	意味
[no%]interval	非推奨。使用しないでください。-xia を使用してください。
[no%]iostream	古い iostream ライブラリ libiostream を使用します [しません]。
[no%]Cstd	C++ 標準ライブラリ libCstd を使用します [しません]。コンパイラ付属の C++ 標準ライブラリヘッダーファイルをインクルードします [しません]。
[no%]Crun	C++ 実行時ライブラリ libCrun を使用します [しません]。
[no%]gc	ガベージコレクション libgc を使用します [しません]。
[no%]stlport4	デフォルトの libCstd の代わりに STLport の標準ライブラリ実装 Version 4.5.3 を使用します [しません]。STLport の実装の詳細は、173 ページの「12.3 STLport」 を参照してください。
[no%]stlport4_dbg	STLport のデバッグ可能なライブラリを使用します [しません]。
[no%]sunperf	Sun Performance Library を使用します [しません]。
[no%]stdcxx4	デフォルトの libCstd の代わりに Apache stdcxx バージョン 4 C++ 標準ライブラリを Solaris で使用します [しません]。このオプションにより、-mt オプションも暗黙的に設定されます。stdcxx ライブラリには、マルチスレッドモードが必要です。このオプションは、コンパイルのたびに、およびアプリケーション全体のリンクコマンドで一貫して使用する必要があります。-library=stdcxx4 を使用してコンパイルされたコードは、デフォルトの -library=Cstd または省略可能な -library=stlport4 を使用してコンパイルされたコードと同じプログラムでは使用できません。
%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=libC が想定されます。
  - library=no%libC で特に除外されないかぎり、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=%none`、`-library=no%Cstd`、`-library=stlport4` のいずれかで特に除外されないかぎり、`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かぎり、`libCrun` ライブラリは常に含まれます。

`-library=%none` が指定されたとしても、標準または互換のどちらのモードであるかに関わりなく、`libm` および `libc` 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 iostream 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=stdcxx4 の場合、Apache stdcxx ライブラリを Solaris OS の /usr/include および /usr/lib にインストールする必要があります。このライブラリは、最近の OpenSolaris リリースでも使用できます。

-library=sunperf と -xlic_lib=sunperf は同じコマンド行で使用できません。

どのコマンド行でも、-library=stlport4、-library=stdcxx4、または -library=Cstd オプションのうち使用できるオプションは、多くても 1 つだけです。

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 または -library=stdcxx4 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 RogueWave ツールライブラリを標準モード (デフォルトモード) で取り込む場合は、libiostream も取り込む必要があります (詳細は、『C++ 移行ガイド』を参照してください)。標準 iostream RogueWave ツールライブラリは、標準モードでのみ使用できます。次のコマンド例は、RogueWave tools.h++ ライブラリオプションの有効もしくは無効な使用法について示します。

```
% CC -compat -library=rwtools7 foo.cc          <-- valid
% CC -compat -library=rwtools7_std foo.cc      <-- invalid

% CC -library=rwtools7,iostream foo.cc         <-- valid, classic iostreams
% CC -library=rwtools7 foo.cc                  <-- invalid

% CC -library=rwtools7_std foo.cc              <-- valid, standard iostreams
% CC -library=rwtools7_std,iostream foo.cc     <-- invalid
```

libCstd と libiostream の両方を含めた場合は、プログラム内で新旧両方の形式の iostream (例: cout と std::cout) を使用して、同じファイルにアクセスしないよう注意してください。同じプログラム内に標準 iostream と従来の iostream が混在し、その両方のコードから同じファイルにアクセスすると、問題が発生する可能性があります。

libC ライブラリをリンクしない互換モードプログラムは、C++ 言語のすべての機能を使用できるわけではありません。同様に、Crun ライブラリ、または Cstd と stlport4 いずれのライブラリもリンクしない標準モードプログラムは、C++ 言語のすべての機能を使用できるわけではありません。

-xnolib を指定すると、-library は無視されます。

警告

別々の手順でコンパイルしてリンクする場合は、コンパイルコマンドに表示される一連の -library オプションをリンクコマンドにも表示する必要があります。

stlport4、Cstd、および iostream のライブラリは、固有の入出力ストリームを実装しています。これらのライブラリの 2 個以上を -library オプションを使って指定した場合、プログラム動作が予期しないものになる恐れがあります。STLport の実装の詳細は、173 ページの「12.3 STLport」を参照してください。

これらのライブラリは安定したものではなく、リリースによって変わることがあります。

関連項目

`-I`、`-l`、`-R`、`-staticlib`、`-xia`、`-xlang`、`-xnolib`、134 ページの「10.4.8 アプリケーションの例」、155 ページの「警告:」、173 ページの「12.3.1 再配布とサポートされる STLport ライブラリ」、『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』、『C++ Interval Arithmetic Programming Reference』

`-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、151 ページの「11.7 C++ 標準ライブラリの置き換え」を参照してください。

A.2.50 -m32|-m64

コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。

`-m32` を使用すると、32 ビット実行可能ファイルと共有ライブラリが作成されます。`-m64` を使用すると、64 ビット実行可能ファイルと共有ライブラリが作成されます。

ILP32 メモリーモデル (32 ビット int、long、ポインタデータ型) は 64 ビット対応ではないすべての Solaris プラットフォームおよび Linux プラットフォームのデフォルトです。LP64 メモリーモデル (64 ビット long、ポインタデータ型) は 64 ビット対応の Linux プラットフォームのデフォルトです。`-m64` は LP64 モデル対応のプラットフォームでのみ使用できます。

`-m32` を使用してコンパイルされたオブジェクトファイルまたはライブラリを、`-m64` を使用してコンパイルされたオブジェクトファイルまたはライブラリにリンクすることはできません。

`-m32|-m64` を指定してコンパイルしたモジュールは、`-m32|-m64` を指定してリンクする必要があります。コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの一覧については、50 ページの「3.3.3 コンパイル時とリンク時のオプション」 を参照してください。

x64 プラットフォームで大量の静的データを持つアプリケーションを `-m64` を使用してコンパイルするときは、`-xmodel=medium` も必要になることがあります。一部の Linux プラットフォームは、ミディアムモデルをサポートしていません。

以前のコンパイラリリースでは、`-xarch` で命令セットを選択すると、メモリーモデル ILP32 または LP64 が使用されていました。Solaris Studio 12 以降のコンパイラでは、このようなことはありません。ほとんどのプラットフォームでは、`-m64` をコマンド行に追加するだけで、64 ビットオブジェクトが作成されます。

Solaris では、-m32 がデフォルトです。64 ビットプログラムをサポートする Linux システムでは、-m64 -xarch=sse2 がデフォルトです。

-xarch も参照してください。

A.2.51 -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。-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。詳細は、mcs(1) のマニュアルページを参照してください。

A.2.52 -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。

A.2.53 -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。次に例を示します。

```
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。

A.2.53.1 相互の関連性

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。

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。

A.2.54 `-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。

A.2.55 `-mt[={yes|no}]`

このオプションを使用して、Solaris スレッドまたは POSIX スレッドの API を使用しているマルチスレッド化コードをコンパイルおよびリンクします。`-mt=yes` オプションにより、ライブラリが適切な順序でリンクされることが保証されます。

このオプションは `-D_REENTRANT` をプリプロセッサに渡します。

Solaris スレッドを使用するには、`thread.h` ヘッダーファイルをインクルードし、`–mt=yes` オプションを使用してコンパイルします。Solaris プラットフォームで POSIX スレッドを使用するには、`pthread.h` ヘッダーファイルをインクルードし、`–mt=yes –lpthread` オプションを使用してコンパイルします。

Linux プラットフォーム上では、POSIX スレッドの API のみが使用できます (Linux プラットフォームには `libthread` はありません)。したがって、Linux プラットフォームで `–mt=yes` を使用すると、`–lthread` の代わりに `–lpthread` が追加されます。Linux プラットフォームで POSIX スレッドを使用するには、`–mt=yes` を使用してコンパイルします。

`–G` を使用してコンパイルする場合は、`–mt=yes` を指定しても、`–lthread` と `–lpthread` のどちらも自動的には含められません。共有ライブラリを構築する場合は、これらのライブラリを明示的にリストする必要があります。

(OpenMP 共有メモリー並列化 API を使用するための) `–xopenmp` オプションには、`–mt=yes` が自動的に含まれます。

-mt=yes を指定してコンパイルを実行し、リンクを個別の手順でリンクする場合は、コンパイル手順と同様にリンク手順でも -mt=yes オプションを使用する必要があります。-mt を使用して 1 つの変換ユニットをコンパイルおよびリンクする場合は、-mt を指定してプログラムのすべてのユニットをコンパイルおよびリンクする必要があります。

-mt=yes は、コンパイラのデフォルトの動作です。この動作が望ましくない場合は、-mt=no でコンパイルします。

オプション –mt は –mt=yes と同じです。

A.2.55.1 関連項目

–xnolib、Solaris 『Multithreaded Programming Guide』、および『Linker and Libraries Guide』

A.2.56 -native

-xtarget=native と同じです。

A.2.57 -noex

-features=no%except と同じです。

A.2.58 -nofstore

x86: 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。

- 式や関数を変数に割り当てる

  または

- 式や関数をそれより短い浮動小数点型にキャストする

A.2.58.1 関連項目

-fstore

A.2.59 -nolib

-xnolib と同じです。

A.2.60 -nolibmil

-xnolibmil と同じです。

A.2.61 -noqueue

(廃止) 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。メイクファイルのテストには、ゼロ以外の状態が返されます。このオプションは廃止されたので無視します。

A.2.62 -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。150 ページの「11.6 共有ライブラリの使用」 を参照してください。

A.2.62.1 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いけません。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。

A.2.63 -O

このリリースから、 -O マクロは、 -xO2 でなく、 -xO3 に展開されます。

このデフォルトの変更によって、実行時のパフォーマンスが向上します。ただし、あらゆる変数を自動的に volatile と見なすことを前提にするプログラムの場合、-xO3 は不適切なことがあります。このことを前提とする代表的なプログラムとしては、専用の同期方式を実装するデバイスドライバや古いマルチスレッドアプリケーションがあります。回避策は、 -O ではなく、 -xO2 を使ってコンパイルすることです。

A.2.64 -O*level*

-O*level*、コンパイラオプション同じです。

A.2.65 -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。

A.2.65.1 相互の関連性

コンパイラは、テンプレートインスタンスを格納する必要がある場合には、出力ファイルのディレクトリにあるテンプレートリポジトリに格納します。たとえば、次のコマンドでは、コンパイラはオブジェクトファイルを `../sub/a.o` に、テンプレートインスタンスを `./sub/SunWS_cache` 内のリポジトリにそれぞれ書き込みます。

```
example% CC -instances=extern -o sub/a.o a.cc
```

コンパイラは、読み込むオブジェクトファイルに対応するテンプレートリポジトリからテンプレートインスタンスを読み取ります。たとえば、次のコマンドは `./sub1/SunWS_Cache` と `./sub2/SunWS_cache` を読み取り、必要な場合は `./SunWS_cache` に書き込みます。

```
example% CC -instances=extern sub1/a.o sub2/b.o
```

詳細は、103 ページの「7.4 テンプレートリポジトリ」を参照してください。

警告

この *filename* 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含むする必要があります。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。

A.2.66 +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。

A.2.66.1 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

A.2.67 -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。

A.2.67.1 関連項目

-E

A.2.68 -p

廃止。342 ページの「A.2.165 -xpg」を参照してください。

A.2.69 -pentium

x86: -xtarget=pentium と置き換えられています。

A.2.70 -pg

-xpg と同じです。

A.2.71 -PIC

SPARC: -xcode=pic32 と同じです。

x86: -Kpic と同じです。

A.2.72 -pic

SPARC: -xcode=pic13 と同じです。

x86: -Kpic と同じです。

A.2.73 -pta

-template=wholeclass と同じです。

A.2.74 -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。

A.2.74.1 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* および105 ページの「7.5.2 定義検索パス」

A.2.75 -pto

-instances=static と同じです。

A.2.76 –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。

A.2.76.1 警告

-ptr オプションは存在しても無視されますが、すべてのコンパイルコマンドから削除するようにしてください。これは将来のリリースで、-ptr が以前とは異なる動作のオプションとして再実装される可能性があるためです。

関連項目

リポジトリのディレクトリについては、103 ページの「7.4 テンプレートリポジトリ」を参照してください。

A.2.77 -ptv

-verbose=template と同じです。

A.2.78 -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。-Q でコンポーネントに渡されるオプションは、順序が変更されることがあります。ドライバが認識するオプションは、正しい順序に保持されます。ドライバがすでに認識しているオプションに、-Q は使わないでください。たとえば C++ コンパイラは、リンカー (ld) に対する -z オプションを認識します。次のようなコマンドを実行したとします。

```
CC -G -zallextract mylib.a -zdefaultextract ... // correct
```

-z オプションは、この順序でリンカーに渡されます。一方、次のようなコマンドを指定したとします。

```
CC -G -Qoption ld -zallextract mylib.a -Qoption ld -zdefaultextract ... // error
```

-z オプションの順序が変わり、不正な結果が生じる可能性があります。

A.2.78.1 値

phase には、次の値のいずれか 1 つを指定します。

表 A–17 -Qoption の値

SPARC	x86
ccfe	ccfe
iropt	iropt
cg	ube
CClink	CClink
ld	ld
—	ir2hf
fbe	fbe

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、次を見てください。

```
-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
```

A.2.79 -qoption *phase option*

`-Qoption` と同じです。

A.2.80 -qp

`-p` と同じです。

A.2.81 -Qproduce *sourcetype*

`CC` ドライバに *sourcetype* (ソースタイプ) 型のソースコードを生成するよう指示します。

sourcetype に指定する接尾辞の定義は次のとおりです。

表 A–18 `-Qproduce` の値

接尾辞	意味
`.i`	`ccfe` が作成する前処理済みの C++ のソースコード
`.o`	コードジェネレータが作成するオブジェクトファイル
`.s`	`cg` が作成するアセンブラソース

A.2.82 -qproduce *sourcetype*

`-Qproduce` と同じです。

A.2.83 -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。

A.2.83.1 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ー命令によって異なります (-xarch を指定しないと、-xarch=generic が想定されます)。

コンパイラが想定するデフォルトのパスを表示するには、–dryrun と –R の各オプションをリンカー ld に渡して出力を検査します。

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。

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』

A.2.84 -readme

-xhelp=readme と同じです。

A.2.85 -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。

A.2.86 -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。

A.2.87 -sb

廃止され、メッセージを表示されずに無視されます。

A.2.88 -sbfast

廃止され、メッセージを表示されずに無視されます。

A.2.89 -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。

A.2.89.1 値

l には、次の値のいずれか 1 つを指定します。

表 A–19 -staticlib の値

値	意味
[no%]*library*	*library* を静的にリンクします [しません]。*library* に有効な値は、-library で有効なすべての値 (%all と %none を除く)、-xlang で有効なすべての値、および (-xia に関連して使用される) interval です。
%all	-library オプション で指定されているすべてのライブラリと、-xlang オプションで指定されているすべてのライブラリ、-xia をコマンド行で指定している場合は区間ライブラリを静的にリンクします。
%none	-library オプションと -xlang オプションに静的に指定されているライブラリをリンクしません。-xia をコマンド行に指定した場合は、区間ライブラリを静的にリンクしません。

デフォルト

-staticlib を指定しないと、-staticlib=%none が想定されます。

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 (correct)
```

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かぎり、libgc はリンクされないためです。

```
example% CC –staticlib=gc (incorrect)
```

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 (correct)
```

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 (incorrect)
```

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 (correct)
```

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 (incorrect)
 This command links the Sun Performance Libraries statically:

example% CC -library=sunperf -staticlib=sunperf (correct)
```

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、あるいは、64 ビットアーキテクチャーのオプションと同等のオプションを使用する場合、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

library に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。

`-xarch=v9`、`-xarch=v9a`、`-xarch=v9b` のいずれか、あるいは、64 ビットアーキテクチャーのオプションと同等のオプションを使用する場合、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。

64 ビット Solaris x86 プラットフォームでは、`-staticlib=Crun` および `-staticlib=Cstd` オプションは機能しません。どのプラットフォームであれ、これらのライブラリを静的にリンクすることは推奨しません。静的リンクすることによって、プログラムが機能しなくなることがあります。

関連項目

`-library`、149 ページの「11.5 標準ライブラリの静的リンク」

A.2.90 -sync_stdio=[yes| no]

C++ の iostream と C の stdio の同期が原因で実行時のパフォーマンスが低下する場合は、このオプションを使用してください。同期が必要なのは、同じプログラム内で iostream を使って cout に書き込み、stdio を使って stdout に書き込みを行う場合だけです。C++ 規格では同期が求められており、このため C++ コンパイラはデフォルトで同期を有効にします。ただし、しばしば、アプリケーションのパフォーマンスは同期なしの方が良くなることがあります。cout と stdout の一方にしか書き込みを行わない場合は、`-sync_stdio=no` オプションを使って同期を無効にすることができます。

A.2.90.1 デフォルト

`-sync_stdio` を指定しなかった場合は、`-sync_stdio=yes` が設定されます。

次に例を示します。

次の例を考えてみましょう。

```
#include <stdio.h>
#include <iostream>
int main()
{
   std::cout << "Hello ";
   printf("beautiful ");
   std::cout << "world!";
   printf("\n");
}
```

同期が有効な場合は、1 行だけ出力されます。

```
Hello beautiful world!
:
```

同期なしの場合、出力が混乱します。

警告

このオプションは、ライブラリではなく実行可能ファイルのリンクでのみ有効です。

A.2.91 -temp=*path*

一時ファイルのディレクトリを定義します。

このオプションは、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。コンパイラは -temp によって設定された値を、TMPDIR の値より優先します。

A.2.91.1 関連項目

-keeptmp

A.2.92 -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。

A.2.92.1 値

opt は次のいずれかの値である必要があります。

表 A–20 -template の値

値	意味
[no%]extdef	別のソースファイルからテンプレート定義を検索します [しません]。no%extdef を使用すると、コンパイラは _TEMPLATE_NO_EXTDEF を事前定義します。
[no%]geninlinefuncs	明示的にインスタンス化されたクラステンプレートのための非参照インラインメンバー関数を生成します [しません]。

表 A–20 -template の値　(続き)

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いけません。そうでない場合は、コンパイラはそのクラスのどのメンバーもインスタンス化しません。

デフォルト

-template オプションを指定しないと、-template=no%wholeclass,extdef が使用されます。

例

次のコードについて考えてみましょう。

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

-template=geninlinefuncs を指定した場合、S の 2 つのメンバー関数は、プログラム内で呼び出されなくてもオブジェクトファイルに生成されます。

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                             [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                             [__1cBS4Ci_Dsmf6F_v_]
```

関連項目

98 ページの「7.2.2 全クラスインスタンス化」、105 ページの「7.5 テンプレート定義の検索」

A.2.93 -time

-xtime と同じです。

A.2.94 -traceback[={ %none|common|*signals_list*}]

実行時に重大エラーが発生した場合にスタックトレースを発行します。

-traceback オプションを指定すると、プログラムによって特定のシグナルが生成された場合に、実行可能ファイルで stderr へのスタックトレースが発行されて、コアダンプが実行され、終了します。複数のスレッドが 1 つのシグナルを生成すると、スタックトレースは最初のスレッドに対してのみ生成されます。

追跡表示を使用するには、リンク時に -traceback オプションをコンパイラコマンド行に追加します。このオプションはコンパイル時にも使用できますが、実行可能バイナリが生成されない場合無視されます。-traceback を -G とともに使用して共有ライブラリを作成すると、エラーが発生します。

表 A–21 -traceback オプション

オプション	意味
common	sigill、sigfpe、sigbus、sigsegv、または sigabrt の共通シグナルのいずれかのセットが発生した場合にスタックトレースを発行することを指定します。
signals_list	スタックトレースを生成するシグナルの名前を小文字で入力してコンマで区切ったリストを指定します。sigquit、sigill、sigtrap、sigabrt、sigemt、sigfpe、sigbus、sigsegv、sigsys、sigxcpu、sigxfsz のシグナル (コアファイルが生成されるシグナル) をキャッチできます。 これらのシグナルの前に no% を付けると、シグナルのキャッチは無効になります。 たとえば、**-traceback=sigsegv,sigfpe** と指定すると、sigsegv または sigfpe が発生した場合にスタックトレースとコアダンプが生成されます。
%none または none	追跡表示を無効にします。

このオプションを指定しない場合、デフォルトは -traceback=%none になります。

= 記号を指定せずに、-traceback だけを指定すると、-traceback=common と同義になります。

注: コアダンプが不要な場合は、次を使用して coredumpsize 制限を 0 に設定できます。

```
% limit coredumpsize 0
```

-traceback オプションは、実行時のパフォーマンスに影響しません。

A.2.95 -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。ほか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。

A.2.95.1 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文では、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 オプションのあとに処理されます。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

A.2.96 -unroll=*n*

-xunroll=*n* と同じです。

A.2.97 -V

-verbose=version と同じです。

A.2.98 -v

-verbose=diags と同じです。

A.2.99 -vdelx

非推奨。使用しないでください。

互換モード (-compat[=4]) のみ

delete[] を使用する式に対し、実行時ライブラリ関数 _vector_delete_ の呼び出しを生成する代わりに _vector_deletex_ の呼び出しを生成します。関数 _vector_delete_ は、削除するポインタおよび各配列要素のサイズという 2 つの引数をとります。

関数 _vector_deletex_ は _vector_delete_ と同じように動作しますが、3 つ目の引数としてそのクラスのデストラクタのアドレスをとります。この引数は Sun 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。

A.2.99.1 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。Sun 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かぎり、このオプションは使用しないでください。

A.2.100 -verbose=*v*[,*v*...]

コンパイラメッセージの詳細度を制御します。

A.2.100.1 値

v には、次に示す値の 1 つを指定します。

表 A–22 -verbose の値

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]。

表 A–22 -verbose の値 (続き)

値	意味
[no%]template	テンプレートインスタンス化 verbose モード (「検証モード」ともいう) を起動します [しません]。verbose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	前述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 を指定されない場合、-verbose=%none が想定されます。

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。

A.2.101 +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

A.2.101.1 デフォルト

+w オプションを指定しない場合、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。

関連項目

–w、 +w2

A.2.102 +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。これは、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。

A.2.102.1 関連項目

+w

A.2.103 -w

ほとんどの警告メッセージを抑止します。

このオプションは、コンパイラが警告を出力しない原因となります。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。

A.2.103.1 関連項目

+w

A.2.104 -Xm

-features=iddollar と同じです。

A.2.105 -xaddr32

(*Solaris x86/x64* のみ) コンパイラフラグ -xaddr32=yes は、結果として生成される実行可能ファイルまたは共有オブジェクトを 32 ビットアドレス空間に制限します。

この方法でコンパイルする実行可能ファイルは、32 ビットアドレス空間に制限されるプロセスを作成する結果になります。

-xaddr32=no を指定する場合は、通常の 64 ビットバイナリが作成されます。

-xaddr32 オプションを指定しないと、**-xaddr32=no** が想定されます。

-xaddr32 だけを指定すると、-xaddr32=yes が想定されます。

このオプションは、-m64 のコンパイルのみ、および SF1_SUNW_ADDR32 ソフトウェア機能をサポートしている Solaris プラットフォームのみに適用できます。Linux カーネルはアドレス空間の制限をサポートしていないので、Linux では、このオプションは使用できません。

単一のオブジェクトファイルが -xaddr32=yes を指定してコンパイルされた場合は、出力ファイル全体が -xaddr32=yes を指定してコンパイルされたものと、リンク時に想定されます。

32 ビットアドレス空間に制限される共有オブジェクトは、制限された 32 ビットモードのアドレス空間内で実行されるプロセスから読み込む必要があります。

詳細は、『Linker and Libraries Guide』で説明されている SF1_SUNW_ADDR32 ソフトウェア機能の定義を参照してください。

A.2.106 -xalias_level[= *n*]

C++ コンパイラで次のコマンドを指定して、型に基づく別名の解析および最適化を実行することができます。

- -xalias_level[=*n*]

  ここで、*n* には any、simple、compatible のいずれかを指定します。

- -xalias_level=any

  このレベルの解析では、ある型を別名で定義できるとものとして処理されます。ただしこの場合でも、一部の最適化が可能です。

- -xalias_level=simple

  単純型は別名で定義されていないものとして処理されます。次の単純型のいずれかの動的な型である記憶オブジェクトの場合を説明します。

char	short int	long int	float
signed char	unsigned short int	unsigned long int	double
unsigned char	int	long long int	long double
wchar_t	unsigned int	unsigned long long int	enumeration types
データポインタ型	関数ポインタ型	データメンバーのポインタ型	関数メンバーのポインタ型

  これらは、次の型の 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、次の型の 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、次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。

A.2.106.1 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、`-xO2` 以下の最適化レベルでは、型に基づく別名の解析および最適化を実行しません。

警告

`reinterpret_cast` またはこれに相当する旧形式のキャストを使用している場合には、解析の前提にプログラムが違反することがあります。また、次の例にあるような共用体の型のパンニングも、解析の前提に違反します。

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

A.2.107 -xannotate[=yes| no]

(*Solaris*) binopt(1) のようなバイナリ変更ツールであとで変換できるバイナリを作成するようコンパイラに指示します。将来のバイナリ解析、コードガバレージ、およびメモリーエラー検出ツールでも、このオプションを使用して構築されたバイナリを使用できます。

これらのツールによるバイナリファイルの変更を防止するには、`-xannotate=no` オプションを使用します。

`-xannotate=yes` オプションを有効にするには、最適化レベルを `-xO1` かそれ以上に設定して使用する必要があります。また、このオプションは、新しいリンカーサポートライブラリインタフェース `-ld_open()` をサポートしているシステムでのみ有

効です。Solaris 9 OS のように、このリンカーインタフェースをサポートしていないオペレーティングシステムでコンパイラを使用している場合は、コンパイラはメッセージを表示せずに -xannotate=no に戻します。新しいリンカーインタフェースは、Solaris 10 patch 127111-07、Solaris 10 Update 5、および OpenSolaris で使用できます。

デフォルトは -xannotate=yes ですが、前述の条件のいずれかが満たされていない場合は、デフォルトは -xannotate=no に戻されます。

Linux プラットフォームでは、このオプションはありません。

A.2.108 -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++ のアーカイブを構築するときには、テンプレートリポジトリ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入れておく必要があります。テンプレートリポジトリは、少なくとも 1 つのオブジェクトファイルを -instances=extern オプションでコンパイルしたときにのみ使用されます。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。

A.2.108.1 値

-xar を指定すると、ar -c -r が起動され、アーカイブがゼロから作成されます。

例

次のコマンド行は、ライブラリファイルとオブジェクトファイルに含まれるテンプレート関数をアーカイブします。

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

警告

テンプレートデータベースの .o ファイルをコマンド行に追加しないでください。

アーカイブを構築するときは、ar コマンドを使用しないでください。CC -xar を使用して、テンプレートのインスタンス化情報が自動的にアーカイブに含まれるようにしてください。

関連項目

ar(1)、表 14–3

A.2.109 -xarch=*isa*

対象となる命令セットアーキテクチャー (*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ー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。

注 – 意図するメモリーモデルとして LP64 (64 ビット) または ILP32 (32 ビット) を指定するには、それぞれ `-m64` または `-m32` オプションを使用してください。次に示すように以前のリリースとの互換性を保つ場合を除いて、`-xarch` オプションでメモリーモデルを指定できなくなりました。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ `-xarch` の値を指定してください。コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧については、50 ページの「3.3.3 コンパイル時とリンク時のオプション」を参照してください。

A.2.109.1 SPARC での -xarch のフラグ

次の表に、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–23 SPARC プラットフォームでの `-xarch` のフラグ

フラグ	意味
`generic`	ほとんどのプロセッサに共通の命令セットを使用します。これはデフォルト値です。
`generic64`	多くのシステムで良好な 64 ビットパフォーマンスを得るためのコンパイルをします(Solaris のみ)。 このオプションは `-m64 -xarch=generic` に相当し、以前のリリースとの互換性のために用意されています。64 ビットでのコンパイルを指定するには、次のものではなく `-m64` を使用してください。 `- xarch=generic64`
`native`	こ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。現在コンパイルしているシステムプロセッサにもっとも適した設定を選択します。
`native64`	こ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 (Solaris のみ)。このオプションは `-m64 -xarch=native` に相当し、以前のリリースとの互換性のために用意されています。

表 A–23 SPARC プラットフォームでの -xarch のフラグ (続き)

フラグ	意味
sparc	SPARC-V9 ISA 用のコンパイルを実行しますが、VIS (Visual Instruction Set) は使用せず、その他の実装に固有の ISA 拡張機能も使用しません。このオプションを使用して、コンパイラは、V9 ISA で良好なパフォーマンスが得られるようにコードを生成できます。
sparcvis	SPARC-V9 + VIS (Visual Instruction Set) version 1.0 + UltraSPARC 拡張機能用のコンパイルを実行します。このオプションを使用すると、コンパイラは、UltraSPARC アーキテクチャー上で良好なパフォーマンスが得られるようにコードを生成することができます。
sparcvis2	UltraSPARC アーキテクチャー + VIS (Visual Instruction Set) version 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。
sparcvis3	SPARC VIS version 3 の SPARC-V9 ISA 用にコンパイルします。SPARC-V9 命令セット、VIS (Visual Instruction Set) version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) version 2.0、積和演算 (FMA) 命令、および VIS (Visual Instruction Set) version 3.0 を含む UltraSPARC-III 拡張機能の命令をコンパイラが使用できるようになります。
sparcfmaf	SPARC-V9 命令セット、VIS (Visual Instruction Set) version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) version 2.0 を含む UltraSPARC-III 拡張機能、および浮動小数点積和演算用の SPARC64 VI 拡張機能の命令をコンパイラが使用できるようになります。 -xarch=sparcfmaf は fma=fused と組み合わせて使用し、ある程度の最適化レベルを指定することで、コンパイラが自動的に積和命令の使用を試みるようにする必要があります。
sparcima	sparcima 版の SPARC-V9 ISA 用にコンパイルします。コンパイラが、SPARC-V9 命令セットに加えて、VIS (Visual Instruction Set) Version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) Version 2.0 を含む UltraSPARC-III 拡張機能、浮動小数点の積和演算用の SPARC64 VI 拡張機能、整数の積和演算用の SPARC64 VII 拡張機能からの命令を使用できるようにします。
v9	-m64 -xarch=sparc に相当します。64 ビットメモリーモデルを得るために -xarch=v9 を使用する古いメイクファイルとスクリプトでは、-m64 だけを使用すれば十分です。
v9a	-m64 -xarch=sparcvis に相当し、以前のリリースとの互換性のために用意されています。
v9b	-m64 -xarch=sparcvis2 に相当し、以前のリリースとの互換性のために用意されています。

また、次のことにも注意してください。

- generic、sparc、sparcvis2、sparcvis3、sparcfmaf、sparcima でコンパイルされたオブジェクトライブラリファイル (.o) をリンクして、一度に実行できます。ただし、実行できるのは、リンクされているすべての命令セットをサポートしているプロセッサのみです。
- 特定の設定で、生成された実行可能ファイルが実行されなかったり、従来のアーキテクチャーよりも実行速度が遅くなったりする場合があります。また、4倍精度 (REAL*16 および long double) 浮動小数点命令は、これらの命令セットアーキテクチャーのいずれにも実装されないため、コンパイラは、そのコンパイラが生成したコードではそれらの命令を使用しません。

A.2.109.2 x86 での -xarch のフラグ

次の表に、x86 プラットフォームでの -xarch フラグを示します。

表 A–24 x86 での -xarch のフラグ

フラグ	意味
amd64	-m64 -xarch=sse2 に相当します (Solaris のみ)。64 ビットメモリーモデルを得るために -xarch=amd64 を使用する古いメイクファイルとスクリプトでは、-m64 だけを使用すれば十分です。
amd64a	-m64 -xarch=sse2a に相当します (Solaris のみ)。
generic	ほとんどのプロセッサに共通の命令セットを使用します。これはデフォルトであり、–m32 でコンパイルする場合の pentium_pro、および –m64 でコンパイルする場合の sse2 に相当します。
generic64	多くのシステムで良好な 64 ビットパフォーマンスを得るためのコンパイルをします(Solaris のみ)。このオプションは -m64 -xarch=generic に相当し、以前のリリースとの互換性のために用意されています。64 ビットでのコンパイルを指定するには、次のものではなく -m64 を使用してください。 - xarch=generic64
native	こ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。現在コンパイルしているシステムプロセッサにもっとも適した設定を選択します。
native64	こ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 (Solaris のみ)。このオプションは -m64 -xarch=native に相当し、以前のリリースとの互換性のために用意されています。
pentium_pro	命令セットを 32 ビット pentium_pro アーキテクチャーに限定します。
pentium_proa	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット pentium_pro アーキテクチャーに追加します。
sse	SSE 命令セットを pentium_pro アーキテクチャーに追加します。

表 A–24 x86 での -xarch のフラグ (続き)

フラグ	意味
ssea	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE アーキテクチャーに追加します。
sse2	SSE2 命令セットを pentium_pro アーキテクチャーに追加します。
sse2a	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE2 アーキテクチャーに追加します。
sse3	SSE3 命令セットを SSE2 命令セットに追加します。
ssse3	SSSE3 命令セットで、pentium_pro、SSE、SSE2、および SSE3 の各命令セットを補足します。
sse4_1	SSSE4.1 命令セットで、pentium_pro、SSE、SSE2、SSE3、および SSSE3 の各命令セットを補足します。
sse4_2	SSSE4.2 命令セットで、pentium_pro、SSE、SSE2、SSE3、SSSE3、および SSSE4.1 の各命令セットを補足します。
amdsse4a	AMD SSE4a 命令セットを使用します。

A.2.109.3 x86 の特記事項

x86 Solaris プラットフォーム用にコンパイルを行う場合に注意が必要な、重要な事項がいくつかあります。

従来の Sun 仕様の並列化プログラムは、x86 では使用できません。代わりに OpenMP を使用してください。従来の並列化命令を OpenMP に変換する方法については、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

-xarch を sse、sse2、sse2a、または sse3 以降に設定してコンパイルしたプログラムは、必ずこれらの拡張子と機能を提供するプラットフォームでのみ実行してください。

Pentium 4 互換プラットフォームの場合、Solaris OS リリースは SSE/SSE2 に対応しています。これより前のバージョンの Solaris OS は SSE/SSE2 に対応していません。-xarch で選択した命令セットが、実行中の Solaris OS で有効ではない場合、コンパイラはその命令セットのコードを生成またはリンクできません。

コンパイルとリンクを別々に行う場合は、必ずコンパイラを使ってリンクし、-xarch 設定で適切な起動ルーチンがリンクされるようにしてください。

x86 の 80 バイト浮動小数点レジスタが原因で、x86 での演算結果が SPARC の結果と異なる数値になる場合があります。この差を最小にするには、 -fstore オプションを使用するか、ハードウェアが SSE2 をサポートしている場合は -xarch=sse2 でコンパイルします。

Solaris と Linux でも、固有の数学ライブラリ (たとえば、sin(x)) が同じではないため、演算結果が異なることがあります。

A.2.109.4 バイナリの互換性の妥当性検査

Solaris Studio 11 と Solaris 10 OS から、これらの特殊化された -xarch ハードウェアフラグを使用してコンパイルし、構築されたプログラムバイナリは、適切なプラットフォームで実行されることが確認されます。

Solaris 10 以前のシステムでは妥当性検査が行われないため、これらのフラグを使用して構築したオブジェクトが適切なハードウェアに配備されることをユーザが確認する必要があります。

これらの -xarch オプションでコンパイルしたプログラムを、適切な機能または命令セット拡張に対応していないプラットフォームで実行すると、セグメント例外や明示的な警告メッセージなしの不正な結果が発生することがあります。

このことは、.il インラインアセンブリ言語関数を使用しているプログラムや、SSE、SSE2、SSE2a、および SSE3 の命令と拡張機能を利用している `__asm()` アセンブラコードにも当てはまります。

A.2.109.5 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 オプションで設定される -xarch のオーバーライドにも使用できます。-xtarget=ultra2 は -xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1 に展開されます。次のコマンドでは、-xarch=v8plusb は、-xtarget=ultra2 の展開で設定された -xarch=v8plusa より優先されます。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

-xarch=generic64、-xarch=native64、-xarch=v9、-xarch=v9a、または -xarch=v9b による -compat[=4] の使用はサポートされていません。

A.2.109.6 警告

このオプションを最適化と併せて使用する場合、適切なアーキテクチャーを選択すると、そのアーキテクチャー上での実行パフォーマンスを向上させることができます。ただし、適切な選択をしなかった場合、パフォーマンスが著しく低下するか、あるいは、作成されたバイナリプログラムが目的のターゲットプラットフォーム上で実行できない可能性があります。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ -xarch の値を指定してください。

A.2.110 -xautopar

注 – このオプションは OpenMP の並列化命令を受け付けません。Sun 固有の MP プラグマは推奨されず、サポートされません。標準命令への移植情報については、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

(SPARC) 複数プロセッサの自動並列化を有効にします。依存性の解析 (ループの繰り返し内部でのデータ依存性の解析) およびループ再構成を実行します。最適化が -xO3 以上でない場合、最適化は -xO3 に引き上げられ、警告が出されます。

独自のスレッド管理を行なっている場合には、-xautopar を使用しないでください。

実行速度を上げるには、マルチプロセッサシステムが必要です。シングルプロセッサシステムでは、通常、生成されたバイナリの実行速度は低下します。

並列化されたプログラムをマルチスレッド環境で実行するには、実行前に OMP_NUM_THREADS 環境変数を設定しておく必要があります。詳細は、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

-xautopar を使用してコンパイルとリンクを 1 度に実行する場合、リンクには自動的にマイクロタスキング・ライブラリおよびスレッドに対して安全な C 実行時ライブラリが含まれます。-xautopar を使用して別々にコンパイルし、リンクする場合、-xautopar でリンクする必要があります。

A.2.110.1 関連項目

334 ページの「A.2.158 -xopenmp[= *i*]」

A.2.111 -xbinopt={prepare| off}

(SPARC) あとでコンパイラ最適化、変換、分析を行うために、バイナリを準備するようコンパイラに命令します。binopt(1) を参照してください。このオプションは、実行可能ファイルまたは共有オブジェクトの構築に使用できます。コンパイルとリンクを別々に行う場合は、両方の手順に -xbinopt を指定する必要があります。

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

一部のソースコードがコンパイルに使用できない場合も、このオプションを使用してそのほかのコードがコンパイルされます。このとき、最終的なバイナリを作成するリンク手順で、このオプションを使用する必要があります。この場合、このオプションでコンパイルされたコードだけが最適化、変換、分析できます。

A.2.111.1 デフォルト

デフォルトは -xbinopt=off です。

相互の関連性

このオプションを有効にするには、最適化レベル -xO1 以上で使用する必要があります。このオプションを使用すると、構築したバイナリのサイズが少し増えます。

-xbinopt=prepare と -g を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

A.2.112 -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。オブジェクトファイル内のコンパイラのコメントからコンパイラが実際に置換を行う関数を特定する方法については、er_src(1) のマニュアルページを参照してください。

-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。

A.2.112.1 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
```

A.2.113 -xcache=*c*

オプティマイザ用のキャッシュ特性を定義します。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指定できますが、-xtarget オプションの拡張機能の一部です。-xtarget オプションによって供給された値をオーバーライドするために使用されます。

このリリースで、キャッシュを共有できるスレッド数を指定するオプションの特性 [/t*i*] が導入されました。

A.2.113.1 値

c には次の値のいずれかを指定します。

表 A–25 -xcache の値

値	意味
generic	これはデフォルトです。ほとんどの SPARC プロセッサに良好なパフォーマンスを提供し、どの x86、SPARC の各プロセッサでも著しいパフォーマンス低下が生じないようなキャッシュ特性を使用するように、コンパイラに指示します。 これらの最高のタイミング特性は、新しいリリースごとに、必要に応じて調整されます。
native	ホスト環境に対して最適化されたパラメータを設定します。
s1/*l1*/*a 1*[/*t*1]	レベル 1 のキャッシュ属性を定義します。
s1/*l1*/*a 1*[/*t*1]:*s2*/*l 2*/*a2*[/*t*2]	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*/*a 1*[/*t*1]:*s2*/*l 2*/*a2*[/*t*2]:*s3*/*l 3*/*a3*[/*t*3]	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*/*ti* 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 バイト単位)
li	レベル *i* のデータキャッシュのラインサイズ (バイト単位)
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 が指定されない場合、デフォルト -xcache=generic が想定されます。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。

t の値を指定しない場合のデフォルトは 1 です。

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	直接マップ結合

関連項目

-xtarget=*t*

A.2.114 -xcg[89|92]

(SPARC) 非推奨、使用しないでください。現在の Solaris オペレーティングシステムのソフトウェアは、SPARC V7 アーキテクチャーをサポートしません。このオプションでコンパイルすると、現在の SPARC プラットフォームでの実行速度が遅いコードが生成されます。-xO を使用して、-xarch、-xchip、および -xcache のコンパイラのデフォルトを利用します。

A.2.115 -xchar[= *o*]

この オプションは、 char 型が符号なしで定義されているシステムからのコード移植を簡単にするためのものです。そのようなシステムからの移植以外では、このオプションは使用しないでください。符号付きまたは符号なしであると明示的に示すように書き直す必要があるのは、符号に依存するコードだけです。

A.2.115.1 値

o には、次のいずれかを指定します。

表 A–26 -xchar の値

値	意味
signed	char 型で定義された文字定数および変数を符号付きとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
s	signed と同義です。
unsigned	char 型で定義された文字定数および変数を符号なしとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
u	unsigned と同義です。

デフォルト

-xchar を指定しない場合は、コンパイラでは -xchar=s が指定されます。

-xchar を値なしで指定した場合は、コンパイラでは -xchar=s が指定されます。

相互の関連性

-xchar オプションは、-xchar でコンパイルしたコードでだけ、char 型の値の範囲を変更します。このオプションは、システムルーチンまたはヘッダーファイル内の char 型の値の範囲は変更されません。特に、CHAR_MAX および CHAR_MIN の値 (limits.h で定義される) は、このオプションを指定しても変更されません。したがって、CHAR_MAX および CHAR_MIN は、通常の char で符号化可能な値の範囲は表示されなくなります。

警告

-xchar=unsigned を使用するときは、マクロでの値が符号付きの場合があるため、char を定義済みのシステムマクロと比較する際には特に注意してください。これは、マクロを使用してエラーコードを戻すルーチンでもっとも一般的です。エ

ラーコードは、一般的には負の値になっています。したがって、char をそのようなマクロによる値と比較するときは、結果は常に false になります。負の数値が符号なしの型の値と等しくなることはありません。

ライブラリを使用してエクスポートしているインタフェース用のルーチンは、-xchar を使用してコンパイルしないようにお勧めします。Solaris ABI では char 型を符号付きとして指定すると、システムライブラリが指定に応じた動作をします。char を符号なしにする影響は、システムライブラリで十分にテストされていませんでした。このオプションを使用しないで、char 型の符号の有無に依存しないようにコードを変更してください。char 型の符号は、コンパイラやオペレーティングシステムによって異なります。

A.2.116 -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 をほか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。

A.2.116.1 値

i には、次のいずれかを指定します。

表 A–27 -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。
init_local	ローカル変数を初期化します。詳細は、『C ユーザーズガイド』を参照してください。
no%init_local	ローカル変数を初期化しません。(デフォルト)

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。

A.2.117 -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。

–xchip オプションは、ターゲットとなるプロセッサを指定してタイミング属性を指定します。このオプションは次のものに影響を与えます。

- 命令の順序 (スケジューリング)
- コンパイラが分岐を使用する方法
- 意味が同じもので代用できる場合に使用する命令

注 – このオプションは単独で指定できますが、 -xtarget オプションの拡張機能の一部です。-xtarget オプションによって供給された値をオーバーライドするために使用されます。

A.2.117.1 値

c には次の値のいずれかを指定します。

表 A–28 -xchip の値

プラットフォーム	値	次の目的のタイミング属性を使用して最適化
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
	native	コンパイルを実行しているシステム上で良好なパフォーマンスを得るため
	old	SuperSPARC プロセッサ より古いプロセッサ
	sparc64vi	SPARC64 VI プロセッサ
	sparc64vii	SPARC64 VII プロセッサ
	super	より古いプロセッサ
	super2	SuperSPARC II プロセッサ
	micro	microSPARC プロセッサ
	micro2	microSPARC II プロセッサ
	hyper	hyperSPARC プロセッサ

表 A–28 -xchip の値 *(続き)*

プラットフォーム	値	次の目的のタイミング属性を使用して最適化
	hyper2	hyperSPARC II プロセッサ
	powerup	Weitek PowerUp プロセッサ
	ultra	UltraSPARC プロセッサ
	ultra2	UltraSPARC II プロセッサ
	ultra2e	UltraSPARC IIe プロセッサ
	ultra2i	UltraSPARC IIi プロセッサ
	ultra3	UltraSPARC III プロセッサ
	ultra3cu	UltraSPARC III Cu プロセッサ
	ultra3i	UltraSparc IIIi プロセッサ
	ultra4	UltraSPARC IV プロセッサ
	ultra4plus	UltraSPARC IVplus プロセッサ
	ultraT1	UltraSPARC T1 プロセッサ.
	ultraT2	UltraSPARC T2 プロセッサ.
	ultraT2plus	UltraSPARC T2+ プロセッサ
	ultraT3	UltraSPARC T3 プロセッサ
x86	generic	大部分の x86 プロセッサ
	core2	Intel Core2 プロセッサ
	nehalem	Intel Nehalem プロセッサ
	opteron	AMD Opteron プロセッサ
	penryn	Intel Penryn プロセッサ
	pentium	Intel Pentium プロセッサ
	pentium_pro	Intel Pentium Pro プロセッサ
	pentium3	Intel Pentium 3 形式プロセッサ
	pentium4	Intel Pentium 4 形式プロセッサ
	amdfam10	AMD AMDFAM10 プロセッサ

デフォルト

ほとんどのプロセッサ上で、generic は、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を使用するようコンパイラに命令するデフォルト値です。

A.2.118 -xcode=*a*

SPARC: コードのアドレス空間を指定します。

注 – 共有オブジェクトの構築では、-xcode=pic13 または -xcode=pic32 を指定することをお勧めします。pic13 または pic32 なしに構築された共有オブジェクトは、正しく機能せず、構築できない可能性があります。

A.2.118.1 値

a には次のいずれかを指定します。

表 A–29 -xcode の値

値	意味
abs32	32 ビット絶対アドレスを生成します。高速ですが範囲が限定されます。コード、データ、および bss を合計したサイズは 2**32 バイトに制限されます。
abs44	SPARC: 44 ビット絶対アドレスを生成します。中程度の速さで中程度の範囲を使用できます。コード、データ、および bss を合計したサイズは 2**44 バイトに制限されます。64 ビットのアーキテクチャーでのみ利用できます。動的 (共有) ライブラリで使用しないでください。
abs64	SPARC: 64 ビット絶対アドレスを生成します。低速ですが範囲は制限されません。64 ビットのアーキテクチャーでのみ利用できます。
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 と同義です。32 ビットアーキテクチャーでは最大 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	位置独立コード (ラージモデル) を生成します。pic13 ほど高速でない可能性がありますが、フルレンジ対応です。-KPIC と同義です。32 ビットアーキテクチャーでは最大 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。

-xcode=pic13 または -xcode=pic32 を使用の決定に際しては、 elfdump -c (詳細は elfdump(1) のマニュアルページを参照) を使用することによって、セクションヘッダー (sh_name: .got) を探して、大域オフセットテーブル (GOT) のサイズを確認してください。sh_size 値が GOT のサイズです。GOT のサイズが 8,192 バイトに満たない場合は -xcode=pic13、そうでない場合は -xcode=pic32 を指定します。

一般に、-xcode の使用方法の決定に際しては、次のガイドラインに従ってください。

- 実行可能ファイルを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 実行可能ファイルへのリンク専用のアーカイブライブラリを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 共有ライブラリを構築する場合は、-xcode=pic13 からスタートし、GOT のサイズが 8,192 バイトを超えたら、-xcode=pic32 を使用する。
- 共有ライブラリへのリンク用のアーカイブライブラリを構築する場合は、-xcode=pic32 のみ使用する。

デフォルト

32 ビットアーキテクチャーの場合は -xcode=abs32 です。64 ビットアーキテクチャーの場合は -xcode=abs44 です。

共有動的ライブラリを作成する場合、64 ビットアーキテクチャーでは -xcode のデフォルト値である abs44 と abs32 を使用できません。-xcode=pic13 または -xcode=pic32 を指定してください。SPARC の場合、-xcode=pic13 および -xcode=pic32 では、わずかですが、次の 2 つのパフォーマンス上の負担がかかります。

- -xcode=pic13 および -xcode=pic32 のいずれかでコンパイルしたルーチンは、共有ライブラリの大域または静的変数へのアクセスに使用されるテーブル (_GLOBAL_OFFSET_TABLE_) を指し示すようレジスタを設定するために、入口で命令を数個余計に実行します。
- 大域または静的変数へのアクセスのたびに、_GLOBAL_OFFSET_TABLE_ を使用した間接メモリー参照が 1 回余計に行われます。-xcode=pic32 でコンパイルした場合は、大域および静的変数への参照ごとに命令が 2 個増えます。

こうした負担があるとしても、-xcode=pic13 あるいは -xcode=pic32 を使用すると、ライブラリコードを共有できるため、必要となるシステムメモリーを大幅に減らすことができます。-xcode=pic13 あるいは -xcode=pic32 でコンパイルした共有ライブラリを使用するすべてのプロセスは、そのライブラリのすべてのコードを共有できます。共有ライブラリ内のコードに非 pic (すなわち、絶対) メモリー参照が 1 つでも含まれている場合、そのコードは共有不可になるため、そのライブラリを使用するプログラムを実行する場合は、その都度、コードのコピーを作成する必要があります。

.o ファイルが -xcode=pic13 または -xcode=pic32 でコンパイルされたかどうかを調べるには、次のように nm コマンドを使用すると便利です。

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

位置独立コードを含む .o ファイルには、_GLOBAL_OFFSET_TABLE_ への未解決の外部参照が含まれます。このことは、英文字の U で示されます。

-xcode=pic13 または -xcode=pic32 を使用すべきかどうかを判断するには、nm を使用して、共有ライブラリで使用または定義されている明確な大域および静的変数の個数を確認します。_GLOBAL_OFFSET_TABLE_ のサイズが 8,192 バイトより小さい場合は、-Kpic を使用できます。そうでない場合は、-xcode=pic32 を使用する必要があります。

A.2.119 -xcrossfile[=*n*]

廃止。使用しないでください。代わりに -xipo を使用してください。

A.2.120 -xdebugformat=[stabs|dwarf]

コンパイラは、デバッグ情報の形式をスタブ形式から「DWARF Debugging Information Format」で規定されている dwarf 形式に変換します。デフォルト設定は -xdebugformat=dwarf です。

デバッグ情報を読み取るソフトウェアを保守している場合は、今回からそのようなツールを stab 形式から dwarf 形式へ移行するためのオプションが加わりました。

このオプションは、ツールを移植する場合に新しい形式を使用する方法として使用してください。デバッガ情報を読み取るソフトウェアを保守していないか、ツールでこれらの内のいずれかの形式のデバッガ情報が必要でなければ、このオプションを使用する必要はありません。

表 A–30 -xdebugformat のフラグ

値	意味
stabs	-xdebugformat=stabs は、stab 標準形式を使用してデバッグ情報を生成します。
dwarf	-xdebugformat=dwarf は、dwarf 標準形式を使用してデバッグ情報を生成します。

-xdebugformat を指定しない場合は、コンパイラでは -xdebugformat=stabs が指定されます。このオプションには引数が必要です。

このオプションは、-g オプションによって記録されるデータの形式に影響します。-g を指定しなくても、一部のデバッグ情報が記録されますが、その情報の形式もこのオプションによって制御されます。したがって、-g を使用しなくても、-xdebugformat は有効です。

dbx とパフォーマンスアナライザソフトウェアは、stab 形式と dwarf 形式を両方とも認識するので、このオプションを使用しても、ツールの機能にはまったく影響を与えません。

注 – これは過渡的なインタフェースなので、今後のリリースでは、マイナーリリースであっても互換性なく変更されることがあります。stab と dwarf のどちらであっても、特定のフィールドまたは値の詳細は、今後とも変更される可能性があります。

詳細は、dumpstabs(1) および dwarfdump(1) のマニュアルページも参照してください。

A.2.121 -xdepend=[yes| no]

(SPARC) ループの繰り返し内部でのデータ依存性の解析およびループ再構成を実行します。この中には、ループ交換、ループ融合、スカラー交換、「デッドアレイ」代入の回避が含まれます。

SPARC では、最適化レベルが –xO3 かそれ以上に設定されている場合はすべて、–xdepend のデフォルトは –xdepend=on です。それ以外の場合は、–xdepend のデフォルトは –xdepend=off です。-xdepend の明示的な設定を指定すると、すべてのデフォルト設定は上書きされます。

x86 では、-xdepend のデフォルトは -xdepend=off です。-xdepend を指定し、最適化が -xO3 以上でない場合は、コンパイラは最適化を -xO3 に上げ、警告を発行します。

引数なしで -xdepend を指定すると、-xdepend=yes と同等であることを意味します。

依存性の解析は -xautopar に含まれています。依存性の解析はコンパイル時に実行されます。

依存性の解析はシングルプロセッサシステムで役立つことがあります。ただし、シングルプロセッサシステムで -xdepend を使用する場合は、-xautopar を指定するべきではありません。-xdepend 最適化は、マルチプロセッサシステム用に実行されるからです。

関連項目: -xprefetch_auto_type

A.2.122 -xdumpmacros[= *value*[, *value*...]]

マクロがプログラム内でどのように動作しているかを調べたいときに、このオプションを使用します。このオプションは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に従って、標準エラー (stderr) に出力されます。-xdumpmacros オプションは、ファイルの終わりまで、または dumpmacros プラグマまたは end_dumpmacros プラグマによって上書きされるまで有効です。376 ページの「B.2.5 #pragma dumpmacro s」 を参照してください。

A.2.122.1 値

value の代わりに次の引数を使用できます。

表 A–31 -xdumpmacros の値

値	意味
[no%]defs	すべての定義済みマクロを出力します [しません]。
[no%]undefs	すべての解除済みマクロを出力します [しません]。
[no%]use	使用されているマクロの情報を出力します [しません]。
[no%]loc	defs、undefs、use の位置 (パス名と行番号) を印刷します [しません]。
[no%]conds	条件付き指令で使用したマクロの使用情報を出力します [しません]。
[no%]sys	システムヘッダーファイルのマクロについて、すべての定義済みマクロ、解除済みマクロ、使用情報を出力します [しません]。
%all	オプションを-xdumpmacros=defs,undefs,use,loc,conds,sys に設定します。この引数は、[no%] 形式の引数と併用すると効果的です。たとえば -xdumpmacros=%all,no%sys は、出力からシステムヘッダーマクロを除外しますが、そのほかのマクロに関する情報は依然として出力します。
%none	あらゆるマクロ情報を出力しません。

オプションの値は追加されていきます。-xdumpmacros=sys -xdumpmacros=undefs を指定した場合と、-xdumpmacros=undefs,sys を指定した場合の効果は同じです。

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果はありません。たとえば -xdumpmacros=loc,conds,sys は、まったく効果を持ちません。

デフォルト

引数を付けないで -xdumpmacros を指定した場合、-xdumpmacros=defs,undefs,sys を指定したことになります。-xdumpmacros を指定しなかった場合、デフォルト値として -xdumpmacros=%none が使用されます。

例

-xdumpmacros=use,no%loc オプションを使用すると、使用した各マクロの名前が一度だけ出力されます。より詳しい情報が必要であれば、-xdumpmacros=use,loc オプションを使用します。マクロを使用するたびに、そのマクロの名前と位置が印刷されます。

次のファイル t.c を考慮します。

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b) a+b
#else
#define COMPUTE(a,b) a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

次の例は、defs、undefs、sys、および loc の引数に基づいた、ファイル t.c の出力を示しています。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_CC 0x590
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_CC 0x590
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
```

```
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #undef BAR
t.c, line 2: #undef FOO
t.c, line 3: #define COMPUTE(a, b) a + b
```

次の例では、use、loc、および conds の引数によって、マクロ動作がファイル t.c に出力されます。

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1: used macro FOO
t.c, line 7: used macro COMPUTE
t.c, line 8: used macro COMPUTE
t.c, line 9: used macro COMPUTE
t.c, line 9: used macro NN
t.c, line 9: used macro MM
```

次は、ファイル y.c の例です。

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

次は、y.c 内のマクロに基づいた -xdumpmacros=use,loc の出力です。

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4: used macro Z
y.c, line 4: used macro Y
y.c, line 4: used macro X
```

関連項目

dumpmacros プラグマと end_dumpmacros プラグマを使用すれば、-xdumpmacros のスコープを変更できます。

A.2.123 -xe

構文エラーと意味エラーの有無チェックのみ行います。-xe を指定すると、オブジェクトコードは出力されません。-xe の出力は、stderr に送られます。

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

A.2.123.1 関連項目

-c

A.2.124 -xF[=*v*[, *v*...]]

リンカーによる関数と変数の最適な順序の並べ替えを有効にします。

このオプションは、関数とデータ変数を細分化された別々のセクションに配置するようコンパイラに指示します。それによってリンカーは、リンカーの -M オプションで指定されたマップファイル内の指示に従ってこれらのセクションの順序を並べ替えて、プログラムのパフォーマンスを最適化することができます。通常は、この最適化によって効果が上がるのは、プログラムの実行時間の多くがページフォルト時間に費やされている場合だけです。

変数を並べ替えることによって、実行時間のパフォーマンスにマイナスの影響を与える次のような問題を解決できます。

- メモリー内で関係ない変数どうしが近接しているために生じる、キャッシュやページの競合
- 関係のある変数がメモリー内で離れた位置にあるために生じる、不必要に大きな作業セットサイズ
- 使用していない weak 変数のコピーが有効なデータ密度を低下させた結果生じる、不必要に大きな作業セットサイズ

最適なパフォーマンスを得るために変数と関数の順序を並べ替えるには、次の処理が必要です。

1. -xF によるコンパイルとリンク
2. 『プログラムのパフォーマンス解析』マニュアル内の、関数のマップファイルを生成する方法に関する指示に従うか、または、『リンカーとライブラリ』内の、データのマップファイルを生成する方法に関する指示に従います。
3. リンカーの -M オプションを使用して新しいマップファイルを再リンクします。
4. アナライザで再実行して、パフォーマンスが向上したかどうかを検証します。

A.2.124.1 値

v には、次のいずれかを指定します。

表 A–32 -xF の値

値	意味
[no%]func	関数を個々のセクションに細分化します [しません]。
[no%]gbldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
[no%]lcldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
%all	関数、大域データ、局所データを細分化します。
%none	何も細分化しません。

デフォルト

-xF を指定しない場合のデフォルトは、-xF=%none です。引数を指定しないで -xF を指定した場合のデフォルトは、-xF=%none,func です。

相互の関連性

-xF=lcldata を指定するとアドレス計算最適化が一部禁止されるので、このフラグは実験として意味があるときにだけ使用するとよいでしょう。

関連項目

analyzer(1)、ld(1) のマニュアルページも参照してください。

A.2.125 -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。

A.2.126 -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。

A.2.127 -xhwcprof

(SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。

-xhwcprof が有効な場合、ツールがプロファイリング済み負荷を関連付け、命令をデータ型および構造メンバーと一緒に (参照先の -g で生成されたシンボリック情報と組み合わせて) 格納するために役立つ情報を、コンパイラが生成します。プロファイルデータは、ターゲットの命令空間ではなく、データ空間と関連付けられ、命令のプロファイリングだけでは入手の容易でない、動作に関する詳細情報が提供されます。

指定した一連のオブジェクトファイルは、-xhwcprof を指定してコンパイルできます。ただし、-xhwcprof がもっとも役に立つのは、アプリケーション内のすべてのオブジェクトファイルに適用したときです。このオプションによって、アプリケーションのオブジェクトファイルに分散しているすべてのメモリー参照を識別したり、関連付けたりするカバレージが提供されます。

コンパイルとリンクを別々に行う場合は、-xhwcprof をリンク時にも使用してください。将来 -xhwcprof に拡張する場合は、リンク時に -xhwcprof を使用する必要があります。

-xhwcprof=enable または -xhwcprof=disable のインスタンスは、同じコマンド行にある以前の -xhwcprof のインスタンスをすべて無効にします。

-xhwcprof はデフォルトでは無効です。引数を指定せずに -xhwcprof と指定することは、-xhwcprof=enable と指定することと同じです。

-xhwcprof では、最適化を有効にして、デバッグのデータ形式を DWARF (-xdebugformat=dwarf) に設定する必要があります。

-xhwcprof と -g を組み合わせて使用すると、コンパイラに必要な一時ファイル記憶領域は、-xhwcprof と -g を単独で指定することによって増える量の合計を超えて大きくなります。

次のコマンドは example.cc をコンパイルし、ハードウェアカウンタによるプロファイリングのサポートを指定し、DWARF シンボルを使用してデータ型と構造体メンバーのシンボリック解析を指定します。

```
example% CC -c -O -xhwcprof -g -xdebugformat=dwarf example.cc
```

ハードウェアカウンタによるプロファイリングの詳細については、『プログラムのパフォーマンス解析』を参照してください。

A.2.128 -xia

区間演算ライブラリをリンクし、適切な浮動小数点環境を設定します。

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。

x86 プラットフォームでは、SSE2 命令セットのサポートが必要です。

A.2.128.1 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。区間演算を使用するようにしていて、`-fsimple` か `-ftrap`、`-fns`、`-library` のどれかを指定して `-xia` による設定を無効にした場合、コンパイラが不正な動作をすることがあります。

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 `iostream` 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

`-library`

A.2.129 `-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 `-xO3` レベル以上でインライン化するかを指定します。

A.2.129.1 値

func_spec には次の値のいずれかを指定します。

表 A–33 -xinline の値

値	意味
%auto	最適化レベル -xO4 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。

-xipo [=1|2] を使用しないかぎり、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、ほか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、そのほかの関数のインライン化を禁止するには次のコマンドを使用します。

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

相互の関連性

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 以上では、-xinline オプションを指定しなくてもオプティマイザでどの関数をインライン化する必要があるかを判断します。-xO4 では、コンパイラはどの関数が、インライン化されたときにパフォーマンスを改善するかを判断しようとします。

ルーチンは、次のいずれかの条件が当てはまる場合はインライン化されます。

- 最適化は -xO3 以上
- インライン化に効果があって安全
- コンパイル中のファイルの中に関数がある、または-xipo[=1|2] を使用してコンパイルしたファイルの中に関数がある

警告

-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

関連項目

319 ページの「A.2.136 -xldscope={*v*}」

A.2.130 -xinstrument=[no%]datarace

スレッドアナライザで分析するためにプログラムをコンパイルして計測するには、このオプションを指定します。スレッドアナライザの詳細については、tha(1) を参照してください。

そうすることで、パフォーマンスアナライザを使用して計測されたプログラムを collect -r races で実行し、データ競合の検出実験を行うことができます。計測されたコードをスタンドアロンで実行できますが、低速で実行されます。

-xinstrument=no%datarace を指定して、スレッドアナライザ用のソースコードの準備をオフにすることができます。これはデフォルト値です。

-xinstrument を引数なしで指定することはできません。

コンパイルとリンクを別々に行う場合は、コンパイル時とリンク時の両方で -xinstrument=datarace を指定する必要があります。

このオプションは、プリプロセッサトークン __THA_NOTIFY を定義します。#ifdef __THA_NOTIFY を指定して、libtha(3) ルーチンへの呼び出しを保護できます。

このオプションにも、-g を設定します。

A.2.131 -xipo[={0|1|2}]

内部手続きの最適化を実行します。

-xipo オプションが内部手続き解析パスを呼び出すことでプログラムの一部の最適化を実行します。このオプションを指定すると、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。ただし、-xipo によるプログラム全体の最適化には、アセンブリ (.s) ソースファイルは含まれません。

-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、オブジェクトファイルまたはライブラリは対象外です。

A.2.131.1 値

-xipo オプションには次の値があります。

表 A–34 -xipo の値

値	意味
0	内部手続きの最適化を実行しません
1	内部手続きの最適化を実行します
2	内部手続きの別名解析と、メモリーの割り当ておよび配置の最適化を実行し、キャッシュ性能を向上します

デフォルト

-xipo を指定しないと、-xipo=0 が使用されます。

-xipoだけを指定すると、-xipo=1 が使用されます。

例

次の例では同じ手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

オプティマイザは 3 つのすべてのソースファイル間でファイル間のインライン化を実行します。ソースファイルのコンパイルをすべて 1 回のコンパイルで実行しないで済むように、またいくつかの個別のコンパイル時にそれぞれ -xipo オプションを指定して行えるように最後のリンク手順でファイル間のインライン化を実行します。

次の例では別々の手順でコンパイルしてリンクします。

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog part1.o part2.o part3.o
```

コンパイルステップで作成されるオブジェクトファイルは、それらのファイル内でコンパイルされる追加の分析情報を保持します。そのため、リンクステップにおいてファイル相互の最適化を実行できます。

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

コンパイルとリンクを個別に実行する場合、 -xipo をコンパイルとリンクの両方で指定しなければなりません。

-xipo なしでコンパイルされたオブジェクトは、 -xipo で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 four.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、相互手続きの最適化は、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 オプションは、ファイルを介して最適化を実行する際に必要な情報を追加するため、非常に大きなオブジェクトファイルを生成します。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、そのほかに最適化を実行したことに起因します。

A.2.131.2 -xipo= を使用しない内部手続き解析を行う場合

内部手続き解析では、コンパイラは、リンクステップでオブジェクトファイル群を操作しながら、プログラム全体の解析と最適化を試みます。このとき、コンパイラは、このオブジェクトファイル群に定義されているすべての foo() 関数 (またはサブルーチン) に関して次の 2 つのことを仮定します。

- 実行時、このオブジェクトファイル群の外部で定義されている別のルーチンによって foo() が明示的に呼び出されない。
- オブジェクトファイル群内のルーチンからの foo() 呼び出しが、そのオブジェクトファイル群の外部に定義されている別のバージョンの foo() によって置き換えられることがない。

アプリケーションに対して仮定 1 が当てはまらない場合は、コンパイルで -xipo=2 を使わないでください。

仮定 2 が当てはまらない場合は、コンパイルで -xipo=1 および -xipo=2 を使わないでください。

1 例として、独自のバージョンの malloc() で関数 malloc() を置き換え、-xipo=2 を指定してコンパイルするケースを考えてみましょう。-xipo=2 を使ってコンパイルする場合、その独自のコードとリンクされる malloc() を参照する、あらゆるライブラリのあらゆる関数のコンパイルで -xipo=2 を使用する必要があるとともに、リンクステップでそれらのオブジェクトファイルが必要になります。システムライブラリでは、このことが不可能なことがあり、このため、独自のバージョンの malloc のコンパイルに -xipo=2 を使ってはいけません。

もう 1 つの例として、別々のソースファイルにある foo() および bar() という 2 つの外部呼び出しを含む共有ライブラリを構築するケースを考えてみましょう。また、bar() は foo() を呼び出すと仮定します。foo() が実行時に割り込みを受ける可能性がある場合、foo() および bar() のソースファイルのコンパイルで -xipo=1 や -xipo=2 を使ってはいけません。さもないと、foo() が bar() 内にインライン化され、不正な結果になる可能性があります。

関連項目

-xjobs

A.2.132 -xipo_archive=[*a*]

新しい -xipo_archive オプションは、実行可能ファイルを生成する前に、-xipo を付けてコンパイルされ、アーカイブライブラリ (.a) の中に存在しているオブジェクトファイルを使用してリンカーに渡すオブジェクトファイルを最適化します。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルはすべて、その最適化されたバージョンに置き換えられます。

a には、次のいずれかを指定します。

表 A–35 -xipo_archive のフラグ

値	意味
writeback	実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルは、すべてその最適化されたバージョンに置き換えられます。 アーカイブライブラリの共通セットを使用する並列リンクでは、最適化されるアーカイブライブラリの独自のコピーを、各リンクでリンク前に作成する必要があります。
readonly	実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。 -xipo_archive=readonly オプションを指定すると、リンク時に指定されたアーカイブライブラリのオブジェクトファイルで、モジュール間のインライン化と内部手続きデータフロー解析が有効になります。ただし、モジュール間のインライン化によってほかのモジュールに挿入されたコード以外のアーカイブライブラリのコードに対する、モジュール間の最適化は有効になりません。 アーカイブライブラリ内のコードにモジュール間の最適化を適用するには、-xipo_archive=writeback を指定する必要があります。これを行うと、コードが抽出されたアーカイブライブラリの内容が変更されることがあります。
none	これはデフォルト値です。アーカイブファイルの処理は行いません。コンパイラは、モジュール間のインライン化やその他のモジュール間の最適化を、-xipo を使用してコンパイルされ、リンク時にアーカイブライブラリから抽出されたオブジェクトファイルに適用しません。これを行うには、-xipo と、-xipo_archive=readonly または -xipo_archive=writeback のいずれかをリンク時に指定する必要があります。

-xipo_archive の値が指定されない場合、-xipo_archive=none に設定されます。

-xipo_archive をフラグなしで指定することはできません。

A.2.133 -xjobs=*n*

コンパイラが処理を行うために生成するプロセスの数を設定するには、-xjobs オプションを指定します。このオプションを使用すると、マルチ CPU マシン上での構築時間を短縮できます。現時点では、-xjobs とともに使用できるのは -xipo オプ

ションだけです。-xjobs=*n* を指定すると、内部手続きオプティマイザは、さまざまなファイルをコンパイルするために呼び出すことができるコードジェネレータインスタンスの最大数として、*n* を使用します。

A.2.133.1 値

-xjobs には必ず値を指定する必要があります。値を指定しないと、エラー診断が表示され、コンパイルは中止します。

一般に、*n* に指定する確実な値は、使用できるプロセッサ数に 1.5 を掛けた数です。生成されたジョブ間のコンテキスト切り替えにより生じるオーバーヘッドのため、使用できるプロセッサ数の何倍もの値を指定すると、パフォーマンスが低下することがあります。また、あまり大きな数を使用すると、スワップ領域などシステムリソースの限界を超える場合があります。

デフォルト

コマンド行に複数の -xjobs のインスタンスがある場合、一番右にあるインスタンスが実行されるまで相互に上書きします。

例

次の例に示すコマンドは 2 つのプロセッサを持つシステム上で、-xjobs オプションを指定しないで実行された同じコマンドよりも高速にコンパイルを実行します。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

A.2.134 -xkeepframe[=[%all,%none,*name*,no% *name*]]

指定した機能 (*name*) のスタック関連の最適化を禁止します。

%all - すべてのコードのスタック関連の最適化を禁止します。

%none - すべてのコードのスタック関連の最適化を許可します。

このオプションがコマンド行で指定されていないと、コンパイラはデフォルトの -xkeepframe=%none を使用します。このオプションが値なしで指定されると、コンパイラは -xkeepframe=%all を使用します。

A.2.135 -xlang=*language*[,*language*]

該当する実行時ライブラリをインクルードし、指定された言語に適切な実行時環境を用意します。

A.2.135.1 値

language は `f77`、`f90`、`f95`、`c99` のいずれかとします。

`f90` 引数と `f95` 引数は同じです。`c99` 引数は、cc `-xc99=%all` を付けてコンパイルされ、CC を付けてリンクされようとしているオブジェクトに対して ISO 9899:1999 C プログラミング言語の動作を呼び出します。

相互の関連性

`-xlang=f90` と `-xlang=f95` の各オプションは `-library=f90` を意味し、`-xlang=f77` オプションは `-library=f77` を意味します。ただし、`-library=f77` と `-library=f90` の各オプションは、`-xlang` オプションしか正しい実行時環境を保証しないので、言語が混合したリンクには不十分です。

言語が混合したリンクの場合、ドライバは次の順序で言語階層を使用してください。

1. C++
2. Fortran 95 (または Fortran 90)
3. FORTRAN 77
4. C または C99

Fortran 95、FORTRAN 77、および C++ のオブジェクトファイルを一緒にリンクする場合は、最上位言語のドライバを使用します。たとえば、C++ と Fortran 95 のオブジェクトファイルをリンクするには、次の C++ コンパイラコマンドを使用してください。

```
example% CC -xlang=f95...
```

Fortran 95 と FORTRAN 77 のオブジェクトファイルをリンクするには、次のように Fortran 95 のドライバを使用します。

```
example% f95 -xlang=f77...
```

`-xlang` オプションと `-xlic_lib` オプションを同じコンパイラコマンドで使用することはできません。`-xlang` を使用していて、しかも Sun Performance Library でリンクする必要がある場合は、代わりに `-library=sunperf` を使用してください。

警告

`-xlang` と一緒に `-xnolib` を使用しないでください。

Fortran 並列オブジェクトを C++ オブジェクトと混合している場合は、リンク行に `-mt` フラグを指定する必要があります。

関連項目

`-library`、`-staticlib`

A.2.136 -xldscope={*v*}

extern シンボルの定義に対するデフォルトのリンカースコープを変更するには、-xldscope オプションを指定します。デフォルトを変更すると、実装がよりうまく隠蔽されるため、共有ライブラリと実行可能ファイルをより高速かつ安全に使用できるようになります。

A.2.136.1 値

v には、次のいずれかを指定します。

表 A–36 -xldscope の値

値	意味
global	大域リンカースコープは、もっとも制限の少ないリンカースコープです。シンボル参照はすべて、そのシンボルが定義されている最初の動的ロードモジュール内の定義と結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。
symbolic	シンボリックリンカースコープは、大域リンカースコープよりも制限されたスコープです。リンク対象の動的ロードモジュール内からのシンボルへの参照はすべて、そのモジュール内に定義されているシンボルと結合します。モジュール外については、シンボルは大域なものとなります。このリンカースコープはリンカーオプション -Bsymbolic に対応します。C++ ライブラリでは -Bsymbolic を使用できませんが、-xldscope=symbolic 指定子は問題なく使用できます。リンカーの詳細については、ld(1) を参照してください。
hidden	隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的ロードモジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。

デフォルト

-xldscope を指定しない場合は、コンパイラでは -xldscope=global が指定されます。値を指定しないで -xldscope を指定すると、コンパイラがエラーを出力します。コマンド行にこのオプションの複数のインスタンスがある場合、一番右にあるインスタンスが実行されるまで前のインスタンスが上書きされていきます。

警告

クライアントがライブラリ内の関数をオーバーライドできるようにする場合は必ず、ライブラリの構築時に関数がインラインで生成されないようにしてください。-xinline を指定して関数名を指定した場合、-xO4 以上でコンパイルした場合 (自動的にインライン化されます)、インライン指定子を使用した場合、または、複数のソースファイルにわたる最適化を使用している場合、コンパイラは関数をインライン化します。

たとえば、ABC というライブラリにデフォルトの allocator 関数があり、ライブラリクライアントがその関数を使用でき、ライブラリの内部でも使用されるものとします。

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

-xO4 以上でライブラリを構築すると、コンパイラはライブラリ構成要素内での ABC_allocator の呼び出しをインライン化します。ライブラリクライアントが ABC_allocator を独自の allocator と置き換える場合、置き換えられた allocator は ABC_allocator を呼び出したライブラリ構成要素内では実行されません。最終的なプログラムには、この関数の相異なるバージョンが含まれることになります。

__hidden 指示子または __symbolic 指示子で宣言されたライブラリ関数は、ライブラリの構築時にインラインで生成することができます。これらの関数がクライアントからオーバーライドされることは、サポートされていません。65 ページの「4.1 リンカースコープ」を参照してください。

__global 指示子で宣言されたライブラリ関数はインラインで宣言しないでください。また、-xinline コンパイラオプションを使用することによってインライン化されることがないようにしてください。

関連項目

-xinline、 -xO、 -xcrossfile

A.2.137 -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。

A.2.137.1 関連項目

数値計算ガイド

A.2.138 -xlibmil

選択された libm ライブラリルーチンを最適化のためにインライン展開します。

注 – このオプションは C++ インライン関数には影響しません。

一部の libm ライブラリルーチンにはインラインテンプレートがあります。このオプションを指定すると、これらのテンプレートが選択され、現在選択されている浮動小数点オプションとプラットフォームに対してもっとも高速な実行可能コードが生成されます。

A.2.138.1 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』

A.2.139 -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。このオプションを使用するときは -fround=nearest を指定することによって、デフォルトの丸めモードを使用する必要があります。

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。

A.2.139.1 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、 -xnolibmopt、 -fround

A.2.140 -xlic_lib=sunperf

非推奨。使用しないでください。代わりに、-library=sunperf を指定します。詳細は、257 ページの「A.2.49 -library=*l*[,*l*...]」を参照してください。

A.2.141 -xlicinfo

このオプションは、コンパイル時には暗黙的に無視されます。

A.2.142 -xlinkopt[= レベル]

オブジェクトファイル内のあらゆる最適化のほかに、結果として出力される実行可能ファイルや動的ライブラリに対してリンク時の最適化も行うようコンパイラに指示します。このような最適化は、リンク時にオブジェクトのバイナリコードを解析することによって実行されます。オブジェクトファイルは書き換えられませんが、最適化された実行可能コードは元のオブジェクトコードとは異なる場合があります。

`-xlinkopt` をリンク時に有効にするには、少なくともコンパイルコマンドで `-xlinkopt` を使用する必要があります。`-xlinkopt` を指定しないでコンパイルされたオブジェクトバイナリについても、オプティマイザは限定的な最適化を実行できます。

`-xlinkopt` は、コンパイラのコマンド行にある静的ライブラリのコードは最適化しますが、コマンド行にある共有 (動的) ライブラリのコードは最適化しません。共有ライブラリを構築する場合 (-G でコンパイルする場合) にも、`-xlinkopt` を使用できます。

A.2.142.1 値

level には、実行する最適化のレベルを 0、1、2 のいずれかで設定します。最適化レベルは、次のとおりです。

表 A–37 `-xlinkopt` の値

値	意味
0	リンクオプティマイザは無効ですこれがデフォルトです。
1	リンク時の命令キャッシュカラーリングと分岐の最適化を含む、制御フロー解析に基づき最適化を実行します。
2	リンク時のデッドコードの除去とアドレス演算の簡素化を含む、追加のデータフロー解析を実行します。

コンパイル手順とリンク手順を別々にコンパイルする場合は、両方の手順に `-xlinkopt` を指定する必要があります。

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

レベルパラメータは、コンパイラのリンク時にだけ使用されます。前述の例では、オブジェクトバイナリが指定された 1 のレベルでコンパイルされていても、リンクオプティマイザレベルは 2 です。

デフォルト

レベルパラメータなしで -xlinkopt を使用することは、-xlinkopt=1 を指定することと同じです。

相互の関連性

このオプションは、プログラム全体のコンパイル時に、プロファイルのフィードバックとともに使用されると、もっとも効果的です。プロファイリングによって、コードでもっともよく使用される部分と、もっとも使用されない部分が明らかになるので、それに基づき処理を集中するよう、構築はオプティマイザに指示します。これは、リンク時に実行されるコードの最適な配置が命令のキャッシュミスを低減できるような、大きなアプリケーションにとって特に重要です。このようなコンパイルの例を次に示します。

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

プロファイルフィードバックの使用方法についての詳細は、350 ページの「A.2.170 –xprofile=*p*」を参照してください。

警告

-xlinkopt を指定してコンパイルする場合は、-zcombreloc リンカーオプションは指定しないでください。

このオプションを指定してコンパイルすると、リンク時間がわずかに増えます。オブジェクトファイルも大きくなりますが、実行可能ファイルのサイズは変わりません。-xlinkopt と -g を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

A.2.143 -xloopinfo

このオプションは、並列化されているループとされていないループを示します。通常は、-xautopar オプションとともに使用します。

A.2.144 -xM

指定した C++ プログラムに対してプリプロセッサだけを実行します。その際、メイクファイル用の依存関係を生成してその結果を標準出力に出力します (make ファイルと依存関係についての詳細は make(1) のマニュアルページを参照してください)。

ただし、-xM を指定すると、インクルードされているヘッダーの依存関係のみを報告し、関連付けられているテンプレート定義ファイルの依存関係を報告しません。メイクファイルの中で .KEEP_STATE 機能を使用して、make ユーティリティが使用する .make.state ファイルの中にあるすべての依存関係を使用することもできます。

A.2.144.1 例

次に例を示します。

```
#include <unistd.h>
void main(void)
{}
```

この例で出力されるものは、次のとおりです。

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

相互の関連性

-xM と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を書き込みます。プリプロセッサがこのファイルへの書き込みを行うたびに、このファイルは上書きされます。

関連項目

メイクファイルおよび依存関係についての詳細は、make(1S) のマニュアルページを参照してください。

A.2.145 -xM1

/usr/include ヘッダーファイルの依存関係とコンパイラで提供されるヘッダーファイルの依存関係を報告しないという点を除くと、-xM と同様にメイクファイルの依存関係を生成します。

-xM1 と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を書き込みます。プリプロセッサがこのファイルへの書き込みを行うたびに、このファイルは上書きされます。

A.2.146 -xMD

-xM と同様にメイクファイルの依存関係を生成しますが、コンパイルを続行します。-xMD は、指定されている場合は -o 出力ファイル名から派生したメイクファイルの依存関係情報の出力ファイルを生成します。または、ファイル名接尾辞を .d に置換 (または追加) して、入力元ファイル名を生成します。-xMD と -xMF を指定する場合、-xMF で指定したファイルに、プリプロセッサはすべてのメイクファイルの依存関係情報を書き込みます。複数のソースファイルで -xMD -xMF または -xMD -o *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

A.2.147 -xMF

メイクファイルの依存関係の出力先ファイルを指定するには、このオプションを使用します。コマンド行で -xMF を使用して、複数の入力ファイルのファイル名を個別に指定することはできません。複数のソースファイルで -xMD -xMF または -xMMD -xMF を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

A.2.148 -xMMD

システムヘッダーファイルを除き、メイクファイルの依存関係を生成するには、このオプションを使用します。これは、-xM1 と同様の機能ですが、コンパイルが続行されます。-xMMD は、指定されている場合は -o 出力ファイル名から派生したメイクファイルの依存関係情報の出力ファイルを生成します。または、ファイル名接尾辞を .d に置換 (または追加) して、入力元ファイル名を生成します。-xMF を指定する場合、コンパイラは代わりに、ユーザーが指定したファイル名を使用します。複数のソースファイルで -xMMD -xMF または -xMMD -o *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

A.2.149 -Merge

SPARC: データセグメントとテキストセグメントをマージします。

オブジェクトファイルのデータは読み取り専用です。また、このデータは ld -N を指定してリンクしないかぎりプロセス間で共有されます。

3 つのオプション **-xMerge -ztext -xprofile=collect** を一緒に使用するべきではありません。**-xMerge** を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 **-ztext** を指定すると、位置に依存するシンボルを読み取り

専用記憶領域内で再配置することを禁止します。**-xprofile=collect** を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

A.2.149.1 関連項目

ld(1) のマニュアルページ

A.2.150 -xmaxopt[=*v*]

このコマンドは、コマンド pragma opt のレベルを指定されたレベルに制限します。*v* は、off、1、2、3、4、5 のいずれかです。デフォルト値は -xmaxopt=off であり、pragma opt は無視されます。引数を指定せずに -xmaxopt を指定すると、-xmaxopt=5 を指定したことになります。

-xO と -xmaxopt の両方を指定する場合、-xO で設定する最適化レベルが -xmaxopt 値を超えてはいけません。

A.2.151 -xmemalign=*ab*

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

想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。*a* (境界整列) と *b* (動作) の両方の値が必要です。*a* は、想定する最大メモリー境界整列です。*b* は、境界整列に失敗したメモリーへのアクセスに対する動作です。

コンパイル時に境界整列が判別できるメモリーへのアクセスの場合、コンパイラはそのデータの境界整列に適したロードおよびストア命令を生成します。

境界整列がコンパイル時に決定できないメモリーアクセスの場合、コンパイラは、境界整列を想定して、必要なロード/ストア命令のシーケンスを生成します。

実行時の実際のデータ境界整列が指定された整列に達しない場合、境界整列に失敗したアクセス (メモリー読み取りまたは書き込み) が行われると、トラップが発生します。このトラップに対して可能な応答は 2 つあります。

- OS がトラップを SIGBUS シグナルに変換します。プログラムがこのシグナルを捕捉しなかった場合、プログラムは異常終了します。プログラムがシグナルを捕捉しても、境界整列に失敗したアクセスが成功するわけではありません。

- 境界整列に失敗したアクセスが正常に成功したかのように OS がアクセスを解釈し、プログラムに制御を戻すことによってトラップを処理します。

A.2.151.1 値

次に、-memalign の境界整列と動作の値を示します。

表 A–38 -xmemalign の境界整列と動作の値

a		*b*	
1	最大 1 バイトの境界整列	i	アクセスを解釈し、実行を継続する
2	最大 2 バイトの境界整列	-s	シグナル SIGBUS を発生させる
4	最大 4 バイトの境界整列	f	-xarch=v9 の不変式の場合にのみ、 4 バイト以下の境界整列に対してシグナル SIGBUS を発生させ、それ以外ではアクセスを解釈して実行を継続する。そのほかすべての -xarch 値では、f フラグは i と同じです。
8	最大 8 バイトの境界整列		
16	最大 16 バイトの境界整列		

b を i か f のいずれかに設定してコンパイルしたオブジェクトファイルにリンクする場合は、必ず、-xmemalign を指定する必要があります。50 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

デフォルト

次のデフォルトの値は、-xmemalign オプションがまったく指定されていない場合にのみ適用されます。

- -xmemalign=8i すべての v8 アーキテクチャーに適用される。
- -xmemalign=8s すべての v9 アーキテクチャーに適用される。

次に、-xmemalign オプションが指定されているが値を持たない場合のデフォルト値を示します。

- -xmemalign=1i すべての -xarch 値に適用される。

例

次の表は、-xmemalign で処理できるさまざまな境界整列の状況とそれに適した -xmemalign 指定を示しています。

表 A–39 -xmemalign の例

コマンド	状況
-xmemalign=1s	すべてのメモリーアクセスの整列が正しくないため、トラップ処理が遅すぎる。
-xmemalign=8i	コード内に境界整列されていないデータへのアクセスが意図的にいくつか含まれているが、それ以外は正しい
-xmemalign=8s	プログラム内に境界整列されていないデータへのアクセスは存在しないと思われる
-xmemalign=2s	奇数バイトへのアクセスが存在しないか検査したい
-xmemalign=2i	奇数バイトへのアクセスが存在しないか検査し、プログラムを実行したい

A.2.152 -xmodel=[*a*]

(x86) -xmodel オプションを使用すると、コンパイラで 64 ビットオブジェクトの形式を Solaris x86 プラットフォーム用に変更できます。このオプションは、そのようなオブジェクトのコンパイル時にのみ指定してください。

このオプションは、64 ビット対応の x64 プロセッサで -m64 も指定した場合にのみ有効です。

a には次のいずれかを指定します。

表 A–40 -xmodel のフラグ

値	意味
small	このオプションは、実行されるコードの仮想アドレスがリンク時にわかっていて、すべてのシンボルが 0 ～ 2^31 - 2^24 - 1 の範囲の仮想アドレスに配置されることがわかっているスモールモデルのコードを生成します。
kernel	すべてのシンボルが 2^64 - 2^31 ～ 2^64 - 2^24 の範囲で定義されるカーネルモデルのコードを生成します。
medium	データセクションへのシンボリック参照の範囲に関する前提がないミディアムモデルのコードを生成します。テキストセクションのサイズとアドレスは、スモールコードモデルの場合と同じように制限されます。静的データが大量にあるアプリケーションでは、-m64 を指定してコンパイルするときに、-xmodel=medium が必要になることがあります。

このオプションは累積的ではないため、コンパイラはコマンド行でもっとも右の -xmodel に従って、モデルの値を設定します。

-xmodel を指定しない場合、コンパイラは -xmodel=small とみなします。引数を指定せずに-xmodel を指定すると、エラーになります。

すべての翻訳単位をこのオプションでコンパイルする必要はありません。アクセスするオブジェクトが範囲内にあれば、選択したファイルをコンパイルできます。

すべての Linux システムが、ミディアムモデルをサポートしているわけではありません。

A.2.153 -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 -lc
```

- 互換モード (-compat)

```
-lC -lm -lc
```

-l オプションの順序は重要です。-lm オプションは -lc の前にある必要があります。

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
```

前述の出力には次の行が含まれます。

```
-lCstd -lCrun -lm -lc
```

A.2.153.1 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ー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。

相互の関連性

-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いけません。

-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。

関連項目

-library、-staticlib、-l

A.2.154 -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。

A.2.155 -xnolibmopt

数学ルーチンのライブラリを使用しません。

A.2.155.1 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、そのあとに使用してください。

```
example% CC –fast –xnolibmopt
```

A.2.156 -xnorunpath

266 ページの「A.2.62 -norunpath」 と同じ

A.2.157 -x0*level*

最適化レベルを指定します。大文字 O のあとに数字の 1、2、3、4、5 のいずれかが続きます。一般的に、プログラムの実行速度は最適化のレベルに依存します。最適化レベルが高いほど、実行時のパフォーマンスは向上します。しかし、最適化レベルが高ければ、それだけコンパイル時間が増え、実行可能ファイルが大きくなる可能性があります。

ごくまれに、-x02 の方がほかの値より実行速度が速くなることがあり、-x03 の方が -x04 より早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、-x0*level* オプションで指定された最適化レベルを使用します。

-x0 には 5 つのレベルがあります。以降では各レベルが SPARC および x86 プラットフォームでどのように動作するかを説明します。

A.2.157.1 値

SPARC プラットフォームの場合

- -x01 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。-x02 や -x03 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 -x01 を使用してください。
- -x02 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。

-x03 では、-x02 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、-x02 を使用してください。一般に、このレベルを使用すると、-xspace オプションと組み合わせないかぎり、コードサイズが大きくなります。

- -x04 では、-x03 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化も自動的に行われます。インライン化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、-xspace オプションと組み合わせないかぎり、コードサイズが大きくなります。
- -x05 では、最高レベルの最適化が行われます。これを使用するのは、コンピュータのもっとも多くの時間を小さなプログラムが使用している場合だけにしてください。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。このレベルの最適化によってパフォーマンスが改善される確率を高くするには、プロファイルのフィードバックを使用します。詳細は、350 ページの「A.2.170 –xprofile=*p*」を参照してください。

x86 プラットフォームの場合

- -x01 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。
- -x02 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。
- -x03 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 で行う最適化レベルに加えて、同じファイルに含まれる関数の自動的なインライン化も行われます。インライン展開を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。このレベルでは一般用のフレームポインタ登録 (edp) も解放します。一般にこのレベルを使用するとコードサイズが大きくなります。
- -xO5 では、最高レベルの最適化が行われます。このレベルで使用される最適化アルゴリズムでは、コンパイル時間が増えたり、実行時間が改善されないことがあります。

相互の関連性

-g または -g0 を使用するとき、最適化レベルが -xO3 以下の場合、最大限のシンボリック情報とほぼ最高の最適化が得られます。

-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 によるデバッグでは、-x0*level* が抑制されませんが、-x0*level* はいくつかの方法で -g を制限します。たとえば、-x0*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。

-xipo オプションは、-xO4 または -xO5 と一緒に使用した場合にのみ効果があります。

-xinline オプションは -xO3 未満の最適化レベルには影響を与えません。-xO4 では、-xinline オプションを指定したかどうかは関係なく、オプティマイザはどの関数をインライン化するかを判断します。-xO4 では、コンパイラはどの関数が、インライン化されたときにパフォーマンスを改善するかを判断しようとします。-xinline を指定して関数のインライン化を強制すると、実際にパフォーマンスを低下させる可能性があります。

デフォルト

デフォルトでは最適化は行われません。ただし、これは最適化レベルを指定しない場合にかぎり有効です。最適化レベルを指定すると、最適化を無効にするオプションはありません。

最適化レベルを設定しないようにする場合は、最適化レベルを示すようなオプションを指定しないようにしてください。たとえば、-fast は最適化を -xO5 に設定するマクロオプションです。それ以外で最適化レベルを指定するすべてのオプションでは、最適化レベルが設定されたという警告メッセージが表示されます。最適化を設定せずにコンパイルする唯一の方法は、コマンド行またはメイクファイルから最適化レ ベルを指定するオプションをすべて削除することです。

警告

大規模な手続き (数千行のコードからなる手続き) に対して `-xO3` または `-xO4` を指定して最適化をすると、不合理な大きさのメモリーが必要になります。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、仮想メモリーを 4G 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 4G
```

このコマンドにより、データ領域が 4G バイトに達したときに、オプティマイザがメモリー不足を回復しようとします。

マシンが使用できるスワップ領域の合計容量を超える値は、制限値として指定することはできません。制限値は、大規模なコンパイル中でもマシンの通常の使用ができるぐらいの大きさにしてください。

最良のデータサイズ設定値は、要求する最適化のレベルと実メモリーの量、仮想メモリーの量によって異なります。

現在のスワップ領域を表示するには次のように入力します。**`swap -l`**

現在の実メモリーを表示するには次のように入力します。**`dmesg | grep mem`**

関連項目

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

A.2.158 -xopenmp[= *i*]

OpenMP 指令による明示的並列化を使用するには、`-xopenmp` オプションを指定します。並列化されたプログラムをマルチスレッド環境で実行するには、実行前に `OMP_NUM_THREADS` 環境変数を設定しておく必要があります。

入れ子並列を有効にするには、`OMP_NESTED` 環境変数を TRUE に設定する必要があります。入れ子並列は、デフォルトでは無効です。

A.2.158.1 値

次の表に *i* の値を示します。

表 A–41　-xopenmp の値

値	意味
parallel	OpenMP プラグマの認識を有効にします。-xopenmp=parallel での最低最適化レベルは -x03 です。コンパイラは必要に応じて低い最適化レベルを -x03 に引き上げ、警告メッセージを表示します。 このフラグは、プロセッサのトークン **_OPENMP** も定義します。
noopt	OpenMP プラグマの認識を有効にします。最適化レベルが -O3 より低い場合は、最適化レベルは上げられません。 CC -O2 -xopenmp=noopt のように、-O3 より低い最適化レベルを明示的に設定した場合、コンパイラはエラーを発生させます。-xopenmp=noopt で最適化レベルを指定しなかった場合、OpenMP プラグマが認識され、その結果プログラムが並列化されますが、最適化は行われません。 このフラグは、プロセッサのトークン **_OPENMP** も定義します。
none	このフラグはデフォルトであり、OpenMP プラグマを認識せず、プログラムの最適化レベルを変更せず、プリプロセッサトークンを事前定義しません。

デフォルト

-xopenmp を指定しない場合は、コンパイラでは -xopenmp=none が指定されます。

-xopenmp は指定されているけれども引数は指定されていない場合、コンパイラはこのオプションを -xopenmp=parallel と設定します。

相互の関連性

dbx を指定して OpenMP プログラムをデバッグする場合、-g と -xopenmp=noopt を指定してコンパイルすれば、並列化部分にブレークポイントを設定して変数の内容を表示することができます。

警告

-xopenmp のデフォルトは、将来変更される可能性があります。警告メッセージを出力しないようにするには、適切な最適化を明示的に指定します。

コンパイルとリンクを別々に実行する場合は、コンパイル手順とリンク手順の両方に -xopenmp を指定してください。共有オブジェクトを作成する場合、このことは重要です。実行可能ファイルのコンパイルに使用されたコンパイラが、-xopenmp を指定して .so を作成するコンパイラよりも古いものであってはいけません。これは、OpenMP 指令を含むライブラリをコンパイルする場合に特に重要です。50 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

最良のパフォーマンスを得るには、OpenMP 実行時ライブラリ `libmtsk.so` の最新パッチが、システムにインストールされていることを確認してください。

関連項目

多重処理アプリケーションを構築するために使用する OpenMP Fortran 95、C、および C++ アプリケーションプログラミングインタフェース (API) の概要については、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

A.2.159 `-xpagesize=`*n*

スタックとヒープの優先ページサイズを設定します。

A.2.159.1 値

次の値は、SPARC で有効です。`4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G`、または `default`。

次の値は、x86/x64 で有効です。`4K`、`2M`、`4M`、`1G`、または `default`。

ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。

Solaris オペレーティングシステムでページのバイト数を調べるには、`getpagesize`(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、`pmap`(1) または `meminfo`(2) を使用します。

デフォルト

`-xpagesize=default` を指定すると、Solaris オペレーティングシステムがページサイズを設定します。

拡張

このオプションは `-xpagesize_heap` と `-xpagesize_stack` のマクロです。これらの 2 つのオプションは `-xpagesize` と同じ次の引数を使用します。`4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G`、`default` のいずれか。両方に同じ値を設定するには `-xpagesize` を指定します。別々の値を指定するには個々に指定します。

警告

`-xpagesize` オプションは、コンパイル時とリンク時に使用しないかぎり無効です。50 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

関連項目

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

A.2.160 -xpagesize_heap=*n*

メモリー上のヒープのページサイズを設定します。

A.2.160.1 値

n の値は、4k、8K、64K、512K、2M、4M、32M、256M、2G、16G、または default です。ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。

Solaris オペレーティングシステムでページのバイト数を調べるには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

デフォルト

-xpagesize_heap=default を指定すると、Solaris オペレーティングシステムはページサイズを設定します。

警告

-xpagesize_heap オプションは、コンパイル時とリンク時に使用しないかぎり無効です。

関連項目

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

A.2.161 -xpagesize_stack=*n*

メモリー上のスタックのページサイズを設定します。

A.2.161.1 値

n の値は、4k、8K、64K、512K、2M、4M、32M、256M、2G、16G、または default です。ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。

Solaris オペレーティングシステムでページのバイト数を調べるには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

デフォルト

-xpagesize_stack=default を指定すると、Solaris オペレーティングシステムはページサイズを設定します。

警告

-xpagesize_stack オプションは、コンパイル時とリンク時に使用しないかぎり無効です。

関連項目

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

A.2.162 -xpch=*v*

このコンパイラオプションは、プリコンパイル済みヘッダー機能を有効にします。プリコンパイル済みヘッダー機能は、ソースファイルが、大量のソースコードを含む一連の共通インクルードファイル群を共有しているようなアプリケーションのコンパイル時間を短縮させることができます。コンパイラは 1 つのソースファイルから一連のヘッダーファイルに関する情報を収集し、そのソースファイルを再コンパイルしたり、同じ一連のヘッダーファイルを持つほかのソースファイルをコンパイルしたりするときにその情報を使用します。コンパイラが収集する情報は、プリコンパイル済みヘッダーファイルに格納されます。この機能を利用するには、-xpch と -xpchstop オプションを、#pragma hdrstop 指令と組み合わせて使用できます。

関連項目

- 341 ページの「A.2.163 -xpchstop=*file*」

■ 378 ページの「B.2.9 #pragma hdrstop」

A.2.162.1 プリコンパイル済みヘッダーファイルの作成

-xpch=*v* を指定する場合、*v* には collect:*pch_filename* または use:*pch_filename* を指定します。-xpch を初回に使用するときは、collect モードを指定する必要があります。-xpch=collect を指定するコンパイルコマンドは、ソースファイルを 1 つしか指定できません。次の例では、-xpch オプションがソースファイル a.cc に基づいて myheader.Cpch というプリコンパイル済みヘッダーファイルを作成します。

```
CC -xpch=collect:myheader a.cc
```

有効なプリコンパイル済みヘッダーファイル名には必ず、.Cpch という接尾辞が付きます。*pch_filename* を指定する場合、自分で接尾辞を追加することも、コンパイラに追加させることもできます。たとえば、cc -xpch=collect:foo a.cc と指定すると、プリコンパイル済みヘッダーファイルには foo.Cpch という名前が付けられます。

プリコンパイル済みヘッダーファイルを作成する場合、プリコンパイル済みヘッダーファイルを使用するすべてのソースファイルで共通な、一連のインクルードファイルを含むソースファイルを選択します。インクルードファイルの並びは、これらのソースファイル全体で同一でなければいけません。collect モードでは、1 つのソースファイル名だけが有効な値である点に注意してください。たとえば、CC -xpch=collect:foo bar.cc は有効ですが、CC -xpch=collect:foo bar.cc foobar.cc は、2 つのソースファイルを指定しているので無効です。

プリコンパイル済みヘッダーファイルの使用方法

プリコンパイル済みヘッダーファイルを使用するには、-xpch=use:*pch_filename* と指定します。プリコンパイル済みヘッダーファイルを作成するために使用されたソースファイルと同じインクルードファイルの並びを持つソースファイルであれば、いくつでも指定できます。たとえば、use モードで、次のようなコマンドがあるとします。CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc.

次の項目が真であれば、既存のプリコンパイル済みヘッダーファイルのみ使用します。次の項目で真ではないものがあれば、プリコンパイル済みヘッダーファイルを再作成する必要があります。

- プリコンパイル済みヘッダーファイルにアクセスするために使用するコンパイラは、プリコンパイル済みヘッダーファイルを作成したコンパイラと同じであること。あるバージョンのコンパイラで作成されたプリコンパイル済みヘッダーファイルは、インストールされているパッチが起因する違いなどから、別のバージョンのコンパイラでは使用できない場合があります。
- -xpch オプション以外で -xpch=use とともに指定するコンパイラオプションは、プリコンパイル済みヘッダーファイルが作成されたときに指定されたオプションと一致すること。

- `-xpch=use` で指定する一連のインクルードヘッダー群は、プリコンパイル済みヘッダーファイルが作成されたときに指定されたヘッダー群と同じであること。
- `-xpch=use` で指定するインクルードヘッダーの内容が、プリコンパイル済みヘッダーファイルが作成されたときに指定されたインクルードヘッダーの内容と同じであること。
- 現在のディレクトリ (すなわち、コンパイルが実行中で指定されたプリコンパイル済みヘッダーファイルを使用しようとしているディレクトリ) が、プリコンパイル済みヘッダーファイルが作成されたディレクトリと同じであること。
- `-xpch=collect` で指定したファイル内の `#include` 指令を含む前処理指令の最初のシーケンスが、`-xpch=use` で指定するファイル内の前処理指令のシーケンスと同じであること。

プリコンパイル済みヘッダーファイルを複数のソースファイル間で共有するために、これらのソースファイルには、最初のトークンの並びとして一連の同じインクルードファイルを使用していなければいけません。この最初のトークンの並びは、活性文字列 (viable prefix) として知られています。活性文字列は、同じプリコンパイル済みヘッダーファイルを使用するすべてのソースファイル間で一貫して解釈される必要があります。

ソースファイルの活性文字列には、コメントと次に示すプリプロセッサ指令のみを指定できます。

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

前述の任意の指令はマクロを参照する場合があります。`#else`、`#elif`、`#endif` 指令は、活性文字列内で一致している必要があります。

プリコンパイル済みヘッダーファイルを共有する各ファイルの活性文字列内では、対応する各 `#define` 指令と `#undef` 指令は同じシンボルを参照する必要があります (`#define` の場合は、各指令は同じ値を参照する必要があります)。各活性文字列内での順序も同じである必要があります。対応する各プラグマも同じで、その順序もプリコンパイル済みヘッダーを共有するすべてのファイルで同じでなければいけません。

プリコンパイル済みヘッダーファイルに組み込まれるヘッダーファイルは、次の項目に違反しないようにしてください。これらの制限に違反するプログラムをコンパイルすると、結果は未定義です。

- ヘッダーファイルには、関数や変数の定義を含めることはできません。
- ヘッダーファイルに、`__DATE__` や `__TIME__` が含まれていてはいけません。これらのプリプロセッサマクロを使用すると、予測できない結果になります。
- ヘッダーファイルに、`#pragma hdrstop`が含まれていてはいけません。

- ヘッダーファイルは、活性文字列内で __LINE__ と __FILE__ を使用できません。インクルードヘッダー内の __LINE__ と __FILE__ は使用できます。

make ファイルの変更方法

-xpch を構築に組み込むためにメイクファイルを変更するには、次の方法があります。

- make と dmake の KEEP_STATE 機能と CCFLAGS 補助変数を使用すれば、暗黙的な make 規則を使用できます。プリコンパイル済みヘッダーが、独立したステップとして出力されます。

```
.KEEP_STATE:
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch: foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out: foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

また、CCFLAGS 補助変数を使用する代わりに、独自のコンパイル規則を定義することもできます。

```
.KEEP_STATE:
.SUFFIXES: .o .cc
%.o:%.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch: foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- KEEP_STATE を使用せずに、通常のコンパイルの二次効果としてプリコンパイル済みヘッダーを生成できますが、それには、明示的なコンパイルコマンドを使用する必要があります。

```
shared.Cpch + foo.o: foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o: ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o: pong.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out: foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

A.2.163 -xpchstop=*file*

-xpchstop=*file* オプションは、-xpch オプションを指定してプリコンパイル済みヘッダーファイルを作成する際の最後のインクルードファイルを指定します。コマ

ンド行で -xpchstop を使用することは、cc コマンドで指定する各ソースファイル内のファイルを参照する最初のインクルード指令のあとに、hdrstop プラグマを配置することと同じです。

次の例では、-xpchstop オプションで、プリコンパイル済みヘッダーファイルの活性文字列が projectheader.h をインクルードして終わるよう指定してます。したがって、privateheader.h は活性文字列の一部ではありません。

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h -c
```

A.2.163.1 関連項目

-xpch、 pragma hdrstop

A.2.164 -xpec[={yes|no}]

移植可能な実行可能コード (Portable Executable Code、PEC) バイナリを生成します。PEC バイナリは、自動チューニングシステム (Automatic Tuning System、ATS) と組み合わせて使用できます。ATS はチューニングとトラブルシューティングの目的で、コンパイル済み PEC バイナリを再構築する方法で動作し、下のソースコードは必要ありません。ATS についての詳細は、http://cooltools.sunsource.net/ats/ (http://cooltools.sunsource.net/ats/) を参照してください。

-xpec で構築したバイナリは通常、-xpec なしで構築したバイナリより 5 〜 10 倍の大きさになります。

-xpec を指定しない場合は、-xpec=no に設定されます。-xpec をフラグなしで指定した場合は、コンパイラでは -xpec=yes が指定されます。

A.2.165 -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します。

-xpg オプションでは、gprof でプロファイル処理するためのデータを収集する自動プロファイルコードをコンパイルします。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。

注 – -xpg を指定した場合、 -xprofile を使用する利点はありません。 これら 2 つは、他方で使用できるデータを生成せず、他方で生成されたデータを使用できません。

プロファイルは、64 ビット Solaris プラットフォームで prof(1) または gprof(1)、32 ビット Solaris プラットフォームで gprof を使用して生成され、おおよそのユーザー CPU 時間が含まれます。これらの時間は、メインの実行可能ファイルのルーチンと、実行可能ファイルをリンクするときにリンカー引数として指定した共有ライブラリのルーチンの PC サンプルデータ (pcsample(2) を参照) から導出されます。そのほかの共有ライブラリ (dlopen(3DL) を使用してプロセスの起動後に開かれたライブラリ) のプロファイルは作成されません。

32 ビット Solaris システムの場合、prof(1) を使用して生成されたプロファイルには、実行可能ファイルのルーチンだけが含まれます。32 ビット共有ライブラリのプロファイルは、 -xpg で実行可能ファイルをリンクし、 gprof(1) を使用することで作成できます。

x86 システムでは、 -xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

Solaris 10 ソフトウェアには、 -p でコンパイルされたシステムライブラリが含まれません。その結果、Solaris 10 プラットフォームで収集されたプロファイルには、システムライブラリルーチンの呼び出し回数が含まれません。

A.2.165.1 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。50 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

関連項目

-xprofile=*p*、analyzer(1) のマニュアルページ、パフォーマンスアナライザのマニュアル

A.2.166 -xport64[=(*v*)]

このオプションを指定すると、64 ビット環境に移植するコードをデバッグできます。このオプションは、具体的には、V8 などの 32 ビットアーキテクチャーを V9 などの 64 ビットアーキテクチャーにコード移植する際によく見られる、型 (ポインタを含む) の切り捨て、符号の拡張、ビット配置の変更といった問題について警告します。

A.2.166.1 値

次に、*v* に指定できる値を示します。

表 A–42 -xport64 の値

値	意味
no	32 ビット環境から 64 ビット環境へのコード移植に関し、まったく警告を生成しない。
implicit	暗黙の変換に関してのみ警告を生成する。明示的なキャストが存在する場合には警告を生成しない。
full	32 ビット環境から 64 ビット環境へのコード移植に関し、あらゆる警告を生成する。具体的には、64 ビット値の切り捨て、ISO 値保護規則に基づく 64 ビットへの符号拡張、ビットフィールドの配置の変更などです。

デフォルト

-xport64 を指定しない場合のデフォルトは、-xport64=no です。-xport64 を値なしで指定した場合は、コンパイラでは -xport64=full が指定されます。

例

以降では、型の切り捨て、符号の拡張、ビット配置の変更を行うコード例を紹介します。

64 ビット値の切り捨てのチェック

V9 などの 64 ビットアーキテクチャーを移植する場合、データが切り捨てられることがあります。切り捨ては、初期化時に代入によって暗黙的に行われることもあれば、明示的なキャストによって行われることもあります。2 つのポインタの違いは typedef ptrdiff_t であり、32 ビットモードでは 32 ビット整数型、64 ビットモードでは 64 ビット整数型です。大きいサイズから小さいサイズの整数型に切り捨てると、次の例にあるような警告が生成されます。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

明示的キャストによってデータが切り捨てられている場合に、64 ビットのパイルモードでの切り捨て警告を抑止するには、-xport64=implicit を使用します。

```
example% CC -c -xarch=v9 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3: Warning: Conversion of 64-bit type value to "int" causes truncation.
1 Warning(s) detected.
example%
```

64 ビットアーキテクチャーへの移植でよく発生するもう 1 つの問題として、ポインタの切り捨てがあります。これは常に、C++ におけるエラーです。切り捨てを引き起こす、ポインタのキャスティングなどの操作は、-xport64 を指定した場合に V9 ではエラー診断となります。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) (((unsigned int)p) & 0xFF); // -xarch=v9 error
  return 0;
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3: Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

符号拡張のチェック

符号なし整数型の式において、通常の ISO C 値保護規則が符号付き整数値の符号拡張に対処している状況があるかどうかを、-xport64 オプションを使用してチェックすることもできます。こういった符号拡張は、実行時に微妙なバグの原因となる可能性があります。

```
example% cat test3.c
int i= -1;
void promo(unsigned long l) {}

int main() {
    unsigned long l;
    l = i;  // warn
    promo(i);       // warn
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6: Warning: Sign extension from "int" to 64-bit integer.
"test3.c", line 7: Warning: Sign extension from "int" to 64-bit integer.
2 Warning(s) detected.
```

ビットフィールドの配置変更のチェック

長いビットフィールドに対する警告を生成するには、-xport64 を使用します。こういったビットフィールドが存在していると、ビットフィールドの配置が大きく変わることがあります。ビットフィールド配置方式に関する前提事項に依存しているプログラムを、64 ビットアーキテクチャーに問題なく移植できるためには、あらかじめ確認作業を行う必要があります。

```
example% cat test4.c
#include <stdio.h>
```

```
union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

V9 における出力

```
example% u.buf[0] = fffffffff000000 u.buf[1] = 0
```

警告

警告が生成されるのは、 `-m64` などのオプションを指定して 64 ビットモードでコンパイルしたときだけです。

関連項目

262 ページの「A.2.50 -m32|-m64」

A.2.167 -xprefetch[=*a*[,*a*...]]

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します。

明示的な先読みは、測定値によってサポートされた特殊な環境でのみ使用すべきです。

a には次のいずれかを指定します。

表 A–43 `-xprefetch` の値

値	意味
`auto`	先読み命令の自動生成を有効にします。
`no%auto`	先読み命令の自動生成を無効にします。
`explicit`	(SPARC) 明示的な先読みマクロを有効にします。
`no%explicit`	(SPARC) 明示的な先読みマクロを無効にします。

表 A–43 -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。このフラグは、-xprefetch=auto とのみ組み合わせることができます。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	廃止。使用しないでください。代わりに - xprefetch=auto,explicit を使用します。
no	廃止。使用しないでください。代わりに -xprefetch=no%auto,no%explicit を使用します。

-xprefetch および -xprefetch=auto を指定すると、コンパイラは生成するコードに自由に先読み命令を挿入します。その結果、先読みをサポートするアーキテクチャーでパフォーマンスが向上します。

大型のマルチプロセッサで集約的なコードを実行する場合、-xprefetch=latx:*factor* を使用すると便利です。このオプションは、指定の係数により、先読みからロードまたはストアまでのデフォルトの応答時間を調整するようにコード生成プログラムに指示します。

先読みの応答時間とは、先読み命令を実行してから先読みされたデータがキャッシュで利用可能となるまでのハードウェアの遅延のことです。コンパイラは、先読み命令と先読みされたデータを使用するロードまたはストア命令の距離を決定する際に先読み応答時間の値を想定します。

注 – 先読みからロードまでのデフォルト応答時間は、先読みからストアまでのデフォルト応答時間と同じでない場合があります。

コンパイラは、幅広いマシンとアプリケーションで最適なパフォーマンスを得られるように先読み機構を調整します。しかし、コンパイラの調整作業が必ずしも最適であるとはかぎりません。メモリーに負担のかかるアプリケーション、特に大型のマルチプロセッサでの実行を意図したアプリケーションの場合、先読みの応答時間の値を引き上げることにより、パフォーマンスを向上できます。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。そのあと、テストの結果に応じて係数を増減し、パフォーマンステストを再実行します。係数の調整を

継続し、最適なパフォーマンスに到達するまでパフォーマンステストを実行します。係数を小刻みに増減すると、しばらくはパフォーマンスに変化がなく、突然変化し、再び平常に戻ります。

A.2.167.1 デフォルト

デフォルトは -xprefetch=auto,explicit です。基本的に非線形のメモリーアクセスパターンを持つアプリケーションには、このデフォルトが良くない影響をもたらします。デフォルトを無効にするには、-xprefetch=no%auto,no%explicit を指定します。

no%auto か no の引数で明示的にオーバーライドされない限り、デフォルト auto が想定されます。たとえば、-xprefetch=explicit は -xprefetch=explicit,auto と同じことです。

デフォルト explicit は、引数に no%explicit か no を指定して明示的に無効にするまで継続されます。たとえば、-xprefetch=auto は -xprefetch=auto,explicit と同じことです。

-xprefetch だけを指定すると、-xprefetch=auto,explicit が使用されます。

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 が想定されます。

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ーが適切なもので、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,explicit、あるいは -xprefetch=explicit が指定されていなければいけません。

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%explicit 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かぎり有効です。つまり、latx:*factor* は、-xprefetch=auto,latx:*factor* とともに使用しないかぎり無視されます。

警告

明示的な先読みは、測定値によってサポートされた特殊な環境でのみ使用すべきです。

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かぎり使用してください。使用先読み応答時間は、リリースごとに変わる可能性があります。したがって、別のリリースに切り替えたら、その都度応答時間係数の影響を再テストすることを推奨します。

A.2.168 -xprefetch_auto_type=*a*

ここで *a* は [no%]indirect_array_access です。

このオプションは、直接メモリーアクセスに対して先読み命令が生成されるのと同じ方法で、-xprefetch_level オプションが指示するループに対して間接先読み命令を生成するかどうかを制御します。

-xprefetch_auto_type の値が指定されていない場合、-xprefetch_auto_type=no%indirect_array_access に設定されます。

-xdepend、-xrestrict、および -xalias_level などのオプションは、メモリー別名を明確化する情報の生成に役立つため、間接先読み候補の計算の攻撃性に影響し、このため、自動的な間接先読みの挿入が促進されることがあります。

A.2.169 -xprefetch_level[=*i*]

-xprefetch_level=*i* オプションを使用して、-xprefetch=auto で決定した先読み命令の自動挿入の攻撃性を調整することができます。-xprefetch_leve 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。

A.2.169.1 値

i には 1、2、3 のいずれかを指定します。

表 A–44 -xprefetch_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。

表 A–44 -xprefetch_level の値　(続き)

値	意味
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=1 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
(v9、v9a、v9b、generic64、native64) でコンパイルした場合にだけ有効です。

A.2.170 –xprofile=*p*

プロファイルのデータを収集したり、プロファイルを使用して最適化したりします。

p には、collect[:*profdir*]、use[:*profdir*]、または tcov[:*profdir*] を指定する必要があります。

このオプションを指定すると、実行頻度のデータが収集されて実行中に保存されます。このデータを以降の実行で使用すると、パフォーマンスを向上させることができます。プロファイルの収集は、マルチスレッド対応のアプリケーションにとって安全です。すなわち、独自のマルチタスク (-mt) を実行するプログラムをプロファイリングすることで、正確な結果が得られます。このオプションは、最適化レベルを -xO2 かそれ以上に指定するときにのみ有効になります。コンパイルとリンクを別々の手順で実行する場合は、リンク手順とコンパイル手順の両方で同じ -xprofile オプションを指定する必要があります。

collect[:*profdir*]　実行頻度のデータを集めて保存します。のちに -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。コンパイラによって文の実行頻度を測定するためのコードが生成されます。

-xMerge、-ztext、および -xprofile=collect を一緒に使用しないでください。-xMerge を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 -ztext を指定すると、位置に依存するシンボルを読み取り専用記憶領域内で再配置することを禁止します。-xprofile=collect を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

プロファイルディレクトリ名として *profdir* を指定すると、この名前が、プロファイル化されたオブジェクトコードを含むプログラムまたは共有ライブラリの実行時にプロファイルデータが保存されるディレクトリのパス名になります。*profdir* パス名が絶対パスではない場合、プログラムがオプション `-xprofile=use:`*profdir* でコンパイルされるときの現在の作業用ディレクトリの相対パスとみなされます。プロファイルディレクトリ名を指定しないと、プロファイルデータは、*program*`.profile` という名前のディレクトリに保存されます (*program* はプロファイル化されたプロセスのメインプログラムのベース名)。

例 [1]: プログラムが構築されたディレクトリと同じディレクトリにある myprof.profile ディレクトリでプロファイルデータを収集して使用するには、次のように指定します。

```
demo: CC -xprofile=collect:myprof.profile -xO5 prog.cc -o prog
demo: ./prog
demo: CC -xprofile=use:myprof.profile -xO5 prog.cc -o prog
```

例 [2]: ディレクトリ `/bench/myprof.profile` にプロファイルデータを収集し、収集したプロファイルデータをあとから最適化レベル `-xO5` のフィードバックコンパイルで使用するには、次のように指定します。

```
demo: CC -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.cc -o prog
...run prog from multiple locations..
demo: CC -xprofile=use:/bench/myprof.profile
\    -xO5 prog.cc -o prog
```

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。これらの環境変数を設定すると、`-xprofile=collect` データが `$SUN_PROFDATA_DIR/$SUN_PROFDATA` に書き込まれます。

これらの環境変数は、`tcov` で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 `tcov`(1) マニュアルページの説明どおり、同様に制御指定します。これらの環境変数をまだ設定していない場合、プロファイルデータは現作業ディレクトリの *profdir* `.profile` に書き込まれます (*profdir* は実行ファイルの名前または `-xprofile=collect:` *profdir* フラグで指定された名前)。*profdir* が `.profile` ですでに終了している場合、`-xprofile` では、`profile` が *profdir* に追加されません。プログラムを複数回実行すると、実行頻度データは *profdir*`.profile` ディレクトリに蓄積されていくので、以前の実行頻度データは失われません。

別々の手順でコンパイルしてリンクする場合は、`-xprofile=collect` を指定してコンパイルしたオブジェクトファイルは、リンクでも必ず `-xprofile=collect` を指定してください。

`use`[`:`*profdir*`]`

`-xprofile=collect`[`:` *profdir*] でコンパイルされたコードから収集された実行頻度データを使用して、プロファイル化されたコードが実行されたときに実行された作業用の最適化が行えます。*profdir* は、`-xprofile=collect`[`:` *profdir*] でコンパイルされたプログラムを実行して収集されたプロファイルデータを含むディレクトリのパス名です。

profdir パス名は省略可能です。 *profdir* が指定されていない場合、実行可能バイナリの名前が使用されます。`-o` が指定されていない場合、`a.out` が使用されます。 *profdir* が指定されていない場合、コンパイラは、`profdir .profile/feedback`、または *a.out.profile/feedback* を探します。次に例を示します。

```
demo: CC -xprofile=collect -o myexe prog.cc
demo: CC -xprofile=use:myexe -xO5 -o myexe    prog.cc
```

`-xprofile=collect` オプションを付けてコンパイルしたときに生成され、プログラムの前の実行で作成されたフィードバックファイルに保存された実行頻度データを使用して、プログラムが最適化されます。

`-xprofile` オプションを除き、ソースファイルおよびコンパイラのほかのオプションは、フィードバックファイルを生成したコンパイル済みプログラムのコンパイルに使用したものと完全に同一のものを指定する必要があります。同じバージョンのコンパイラは、収集構築と使用構築の両方に使用する必要があります。

`-xprofile=collect:`*profdir* を付けてコンパイルした場合は、*-xprofile=use:* `profdir` のコンパイルの最適化に同じプロファイルディレクトリ名 *profdir* を使用する必要があります。

収集 (collect) 段階と使用 (use) 段階の間のコンパイル速度を高める方法については、`-xprofile_ircache` も参照してください。

`tcov`[`:`*profdir*`]`

`tcov`(1) を使用する基本のブロックカバレッジ分析用の命令オブジェクトファイル。

オプションの *profdir* 引数を指定すると、コンパイラは指定された場所にプロファイルディレクトリを作成します。プロファイルディレクトリに保存されたデータは、`tcov`(1) または `-xprofile=use:`*profdir* を付けたコンパイラで使用できます。オ

プションの *profdir* パス名を省略すると、プロファイル化されたプログラムの実行時にプロファイルディレクトリが作成されます。プロファイルディレクトリに保存されたデータは、tcov(1) でのみ使用できます。プロファイルディレクトリの場所は、環境変数 SUN_PROFDATA および SUN_PROFDATA_DIR を使用して指定できます。

profdir で指定された場所が絶対パス名ではない場合、コンパイル時に、現在のオブジェクトファイルが作成されたディレクトリの相対パスとみなされます。いずれかのオブジェクトファイルに *profdir* を指定する場合は、同じプログラムのすべてのオブジェクトファイルに対して同じ場所を指定する必要があります。場所が *profdir* で指定されているディレクトリには、プロファイル化されたプログラムを実行するときにすべてのマシンからアクセスできる必要があります。プロファイルディレクトリはその内容が必要なくなるまで削除できません。コンパイラでプロファイルディレクトリに保存されたデータは、再コンパイルする以外復元できません。

例 [1]: 1 つ以上のプログラムのオブジェクトファイルが -xprofile=tcov:/test/profdata でコンパイルされる場合、/test/profdata.profile という名前のディレクトリがコンパイラによって作成されて、プロファイル化されたオブジェクトファイルを表すデータの保存に使用されます。実行時に同じディレクトリを使用して、プロファイル化されたオブジェクトファイルに関連付けられた実行データを保存できます。

例 [2]: myprog という名前のプログラムが -xprofile=tcov でコンパイルされ、ディレクトリ /home/joe で実行されると、実行時にディレクトリ /home/joe/myprog.profile が作成されて、実行時プロファイルデータの保存に使用されます。

A.2.171 -xprofile_ircache[=*path*]

(SPARC) collect 段階で保存されたコンパイルデータを再利用して use 段階のコンパイル時間を向上させるには、-xprofile=collect|use で -xprofile_ircache[=*path*] を使用します。

大きなプログラムでは、中間データが保存されるため、use 段階のコンパイル時間の効率を大幅に向上させることができます。保存されたデータが必要なディスク容量を相当増やすことがある点に注意してください。

-xprofile_ircache[=*path*] を使用すると、*path* はキャッシュファイルが保存されているディレクトリを上書きします。デフォルトでは、これらのファイルはオブジェク

トファイルと同じディレクトリに保存されます。collect と use 段階が 2 つの別のディレクトリで実行される場合は、パスを指定しておくと便利です。一般的なコマンドシーケンスを次に示します。

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

A.2.172 -xprofile_pathmap

(SPARC) -xprofile=use コマンドも指定する場合は、-xprofile_pathmap=*collect_prefix*:*use_prefix* オプションを使用します。次の項目がともに真で、コンパイラが -xprofile=use でコンパイルされたオブジェクトファイルのプロファイルデータを見つけられない場合は、-xprofile_pathmap を使用します。

- 前回オブジェクトファイルが -xprofile=collect でコンパイルされたディレクトリとは異なるディレクトリで、オブジェクトファイルを -xprofile=use を指定してコンパイルしている。
- オブジェクトファイルはプロファイルで共通ベース名を共有しているが、異なるディレクトリのそれぞれの位置で相互に識別されている。

collect-prefix は、オブジェクトファイルが -xprofile=collect でコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

use-prefix は、オブジェクトファイルが -xprofile=use を指定してコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

-xprofile_pathmap の複数のインスタンスを指定すると、コンパイラは指定した順序でインスタンスを処理します。*-xprofile_pathmap* のインスタンスで指定された各 use-prefix は、一致する *use-prefix* が識別されるか、最後に指定された *use-prefix* がオブジェクトファイルのパス名と一致しないことが確認されるまで、オブジェクトファイルのパス名と比較されます。

A.2.173 -xreduction

自動的な並列化を実行するときにループの縮約を解析します。このオプションは、-xautopar が指定する場合のみ有効です。それ以外の場合は、コンパイラは警告を発行します。

縮約の認識が有効な場合、コンパイラは内積、最大値発見、最小値発見などの縮約を並列化します。これらの縮約によって非並列化コードによる四捨五入の結果と異なります。

A.2.174 -xregs=*r*[,*r*...]

生成コード用のレジスタの使用法を指定します。

r には、appl、float、frameptr サブオプションのいずれか 1 つ以上をコンマで区切って指定します。

サブオプションの前に no% を付けるとそのサブオプションは無効になります。

–xregs サブオプションは、特定のハードウェアプラットフォームでしか使用できません。

例: -xregs=appl,no%float

表 A–45 -xregs サブオプション

値	意味
appl	(SPARC) コンパイラがアプリケーションレジスタをスクラッチレジスタとして使用してコードを生成することを許可します。アプリケーションレジスタは次のとおりです。 g2、g3、g4 (32 ビットプラットフォーム) g2、g3 (64 ビットプラットフォーム) すべてのシステムソフトウェアおよびライブラリは、-xregs=no%appl を指定してコンパイルすることをお勧めします。システムソフトウェア (共有ライブラリを含む) は、アプリケーション用のレジスタの値を保持する必要があります。これらの値は、コンパイルシステムによって制御されるもので、アプリケーション全体で整合性が確保されている必要があります。 SPARC 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ロードおよびストア命令が少なくてすむため、パフォーマンスが向上します。ただし、アセンブリコードで記述された古いライブラリプログラムとの間で衝突が起きることがあります。
float	(SPARC) コンパイラが浮動小数点レジスタを整数値用のスクラッチレジスタとして使用してコードを生成することを許可します。浮動小数点値を使用する場合は、このオプションとは関係なくこれらのレジスタを使用します。浮動小数点レジスタに対するすべての参照をコードから排除する場合は、-xregs=no%float を使用するとともに、決して浮動小数点型をコードで使わないようにする必要があります。

表 A–45 -xregs サブオプション (続き)

値	意味
frameptr	(x86) フレームポインタレジスタ (IA32 の場合 %ebp、AMD64 の場合 %rbp) を汎用レジスタとして使用することを許可します。 デフォルトは -xregs=no%frameptr です。 –features=no%except によって例外も無効になっているのでなければ、C++ コンパイラは –xregs=frameptr を無視します。–xregs=frameptr は –fast の一部ですが、–features=no%except も指定されているのでなければ C++ コンパイラによって無視される点に注意してください。 -xregs=framptr を使用すると、コンパイラは浮動小数点レジスタを自由に使用できるので、プログラムのパフォーマンスが向上します。ただし、この結果としてデバッガおよびパフォーマンス測定ツールの一部の機能が制限される場合があります。スタックトレース、デバッガ、およびパフォーマンスアナライザは、–xregs=frameptr を使用してコンパイルされた機能についてレポートできません。 さらに、Posix pthread_cancel() の C++ 呼び出しは、クリーンアップハンドラの検索に失敗します。 C、Fortran、C++ が混在しているコードで、C または Fortran 関数から直接または間接的に呼び出された C++ 関数が例外をスローする可能性がある場合、このコードは –xregs=frameptr でコンパイルできません。このような言語が混在するソースコードを –fast でコンパイルする場合は、コマンド行の –fast オプションのあとに –xregs=no%frameptr を追加します。 64 ビットのプラットフォームで使用できる多くのレジスタでは、–xregs=frameptr でコンパイルすると、64 ビットコードよりも 32 ビットコードのパフォーマンスが向上する可能性が高くなります。 -xpg も指定されている場合、コンパイラは -xregs=frameptr を無視し、警告を表示します。

SPARC のデフォルトは -xregs=appl,float です。

x86 のデフォルトは -xregs=no%frameptr です。

x86 システムでは、-xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

アプリケーションにリンクする共有ライブラリ用のコードは、-xregs=no%appl,float を指定してコンパイルすることをお勧めします。少なくとも、共有ライブラリとリ

ンクするアプリケーションがこれらのレジスタの割り当てを認識するように、共有ライブラリがアプリケーションレジスタを使用する方法を明示的に示す必要があります。

たとえば、大局的な方法で (重要なデータ構造体を示すためにレジスタを使用するなど) レジスタを使用するアプリケーションは、ライブラリと確実にリンクするため、`-xregs=no%appl` なしでコンパイルされたコードを含むライブラリがアプリケーションレジスタをどのように使用するかを正確に特定する必要があります。

A.2.175 -xrestrict[=*f*]

ポインタ型の値をとる関数の引数を制限付きポインタとして扱います。*f* には次の値のいずれかを指定します。

表 A–46 `-xrestrict` の値

値	意味
`%all`	ファイル全体のすべてのポインタ型引数を制限付きとして扱います。
`%none`	ファイル内のどのポインタ型引数も制限付きとして扱いません。
`%source`	メインソースファイル内に定義されている関数のみ制限付きにします。インクルードファイル内に定義されている関数は制限付きにしません。
fn[,*fn*...]	1 つ以上の関数名のコンマ区切りのリスト。関数リストが指定された場合は、指定された関数内のポインタ型引数を制限付きとして扱います。詳細については、次の358 ページの「A.2.175.1 制限付きポインタ」を参照してください。

このコマンド行オプションは独立して使用できますが、最適化時に使用するのがもっとも適しています。たとえば、次のようなコマンドを使用します。

```
%CC -xO3 -xrestrict=%all prog.cc
```

このコマンドでは、ファイル `prog.c` 内のすべてのポインタ引数を制限付きポインタとして扱います。次のようなコマンド行があるとします。

```
%CC -xO3 -xrestrict=agc prog.cc
```

このコマンドでは、ファイル `prog.c` 内の関数 `agc` のすべてのポインタ引数を制限付きポインタとして扱います。

C プログラミング言語の C99 標準には `restrict` キーワードが導入されましたが、このキーワードは最新の C++ 標準に含まれない点に注意してください。一部のコンパイラには、C99 `restrict` キーワードの C++ 言語拡張があり、`__restrict` または

__restrict__ と表記されることがありますが、Solaris Studio C++ コンパイラには現在のところこの拡張はありません。-xrestrict オプションは、ソースコードで restrict キーワードを部分的に置き換えます。このキーワードを使用しても、関数のすべてのポインタ引数を restrict と宣言する必要があるわけではありません。このキーワードは、主に最適化の機会に影響を与え、関数に渡すことができる引数を制限します。ソースコードから restict または __restrict のすべてのインスタンスを削除しても、プログラムの見た目の動作には影響しません。

デフォルトは %none で、-xrestrict と指定すると -xrestrict=%sourcel と指定した場合と同様の結果が得られます。

A.2.175.1 制限付きポインタ

コンパイラが効率よくループを並列化できるようにするには、左辺値が記憶領域の特定の領域を示している必要があります。別名とは、記憶領域の決まった位置を示していない左辺値のことです。オブジェクトへの 2 個のポインタが別名であるかどうかを判断することは困難です。これを判断するにはプログラム全体を解析することが必要であるため、非常に時間がかかります。次の関数 vsq() を考えてみましょう。

例 A–3 2個のポインタを使用したループ

```
extern "C"
void vsq(int n, double *a, double *b) {
    int i;
    for (i=0; i<n; i++) {
            b[i] = a[i] * a[i];
    }
}
```

ポインタ a および b が異なるオブジェクトをアクセスすることをコンパイラが知っている場合には、ループ内の異なる繰り返しを並列に実行することができます。しかし、ポインタ a および b でアクセスされるオブジェクトが重なりあっていれば、ループを安全に並列実行できなくなります。

コンパイル時に関数 vsq() を単純に解析するだけでは、a および b によるオブジェクトのアクセスが重なりあっているかどうかを知ることはできません。この情報を得るには、プログラム全体を解析することが必要になります。次のコマンド行オプションを使用することにより、ポインタ値の関数パラメータを制限付きポインタとして扱うよう指定できます。-xrestrict[=*func1,…,funcn*] 関数リストを指定する場合は、指定した関数内のポインタパラメータは制限付きとして扱われます。それ以外の場合は、ソースファイル全体の中にあるすべてのポインタパラメータが制限つきとして扱われます (推奨しない)。たとえば、-xrestrict=vsq を使用すると、関数 vsq() についての例では、ポインタ a および b が修飾されます。

ポインタ引数を制限付きとして宣言する場合は、ポインタが個別のオブジェクトを指定すると明示することになります。コンパイラは、a および b が個別の記憶領域を指していると想定します。この別名情報によって、コンパイラはループの並列化を実行することができます。

-xrestrict を正しく使用することはとても重要です。区別できないオブジェクトを指しているポインタを制限付きポインタにしてしまうと、ループを正しく並列化できなくなり、不定な動作をすることになります。たとえば、関数 vsq() のポインタ a および b が重なりあっているオブジェクトを指している場合には、b[i] と a[i+1] などが同じオブジェクトである可能性があります。このとき a および b が制限付きポインタとして宣言されていなければ、ループは順次実行されます。a および b が間違って制限付きであると宣言されていれば、コンパイラはループを並列実行するようになりますが、この場合 b[i+1] の結果は b[i] を計算したあとでなければ得られないので、安全に実行することはできません。

A.2.176 -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。

このオプションを指定すると、すべてのデバッグ情報が実行可能ファイルにコピーされます。dbx のパフォーマンスやプログラムの実行時のパフォーマンスにはあまり影響はありませんが、より多くのディスク容量が必要となります。

このオプションは、-xdebugformat=stabs で指定した場合だけ影響があります。この場合のデフォルトは、デバッグデータを実行可能ファイルにコピーしません。デフォルトのデバッグ形式である -xdebugformat=dwarf を使用する場合は、デバッグデータは常に実行可能ファイルにコピーされます。コピーを防止するオプションはありません。

A.2.177 -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。

A.2.177.1 相互の関連性

このオプションは、最適化レベルの -xO5 と、次のいずれかの値の -xarch を組み合わせた場合にだけ有効です。m32 と m64 の両方で sparc、sparcvis、-sparcvis2、または -sparcvis3。

A.2.177.2 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。例外条件の処理にメモリーベースのトラップを明示的に使用するプログラムでは、このオプションは使用しないでください。

A.2.178 -xsb

非推奨、使用しないでください。ソースブラウザ機能は廃止されました。このオプションはメッセージを表示せずに無視されます。

A.2.179 -xsbfast

非推奨、使用しないでください。ソースブラウザ機能は廃止されました。このオプションはメッセージを表示せずに無視されます。

A.2.180 -xspace

SPARC: コードサイズが大きくなるような最適化を行いません。

A.2.181 -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が重要な場合は、対象となるハードウェアを正確に指定してください。これは特に、新しい SPARC プロセッサ上でプログラムを実行する場合に当てはまります。しかし、ほとんどのプログラムおよび廃止の 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。

t には次の値のいずれかを指定します。`native`、`generic`、`native64`、`generic64`、*system-name*。

`-xtarget` に指定する値は、`-xarch`、`-xchip`、`-xcache` の各オプションの値に展開されます。実行中のシステムで `-xtarget=native` の展開を調べるには、`-xdryrun` コマンドを使用します。

たとえば、-xtarget=ultraT2 は次のものと同等です: -xarch=sparcvis2 -xchip=ultraT2 -xcache=8/16/4:4096/64/16

注 – 特定のホストプラットフォームで -xtarget を展開した場合、そのプラットフォームでコンパイルすると -xtarget=native と同じ -xarch、-xchip、または -xcache 設定にならない場合があります。

A.2.181.1 プラットフォームごとの -xtarget の値

この節では、–xtarget 値をプラットフォームごとに説明します。次の表は、すべてのプラットフォーム向けの –xtarget の値を一覧表示します。

表 A–47 すべてのプラットフォームでの -xtarget の値

値	意味
native	ホストシステムで最高のパフォーマンスが得られます。コンパイラは、ホストシステム用に最適化されたコードを生成します。コンパイラは自身が動作しているマシンで利用できるアーキテクチャー、チップ、キャッシュ特性を判定します。
native64	ホストシステムで 64 ビットのオブジェクトバイナリの最高のパフォーマンスが得られます。コンパイラは、ホストシステム用に最適化された 64 ビットのオブジェクトバイナリを生成します。コンパイラが動作しているマシンで使用できる 64 ビットのアーキテクチャー、チップ、キャッシュ特性を判定します。
generic	これはデフォルト値です。汎用アーキテクチャー、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。
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。
システム名	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。 対象となる実際のシステムを表すシステム名を、次のリストから選択してください。

SPARC プラットフォームの -xtarget の値

SPARC または UltraSPARC V9 での 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。-xtarget を指定し、native64 または generic64 以外のフラグを付ける場合は、-m64 オプションも次のように指定する必要があります。-xtarget=ultra ... -m64。この指定を行わない場合は、コンパイラは 32 ビットメモリーモデルを使用します。

表 A–48 SPARC アーキテクチャーでの -xtarget の展開

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
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	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4:2048/64/4:32768/64/4
ultraT1	sparcvis2	ultraT1	8/16/4/4:3072/64/12/32
ultraT2	sparc	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
ultraT3	sparcvis3	ultraT3	8/16/4:6144/64/24
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10

UltraSPARC IVplus、UltraSPARC T1、および UltraSPARC T2 チップのキャッシュプロパティーについての詳細は、294 ページの「A.2.113 -xcache=c」 を参照してください。

x86 プラットフォームの -xtarget の値

64 ビット x86 プラットフォームでの 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。-xtarget を指定し、native64 または generic64 以外のフラグを付ける場合は、-m64 オプションも次のように指定する必要があります。-xtarget=opteron ... -m64。この指定を行わない場合は、コンパイラは 32 ビットメモリーモデルを使用します。

表 A–49 x86 プラットフォームでの -xtarget の値

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
opteron	sse2	opteron	64/64/2:1024/64/16
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic
pentium3	sse	pentium3	16/32/4:256/32/4
pentium4	sse2	pentium4	8/64/4:256/128/8
nehalem	sse4_2	nehalem	32/64/8:256/64/8:8192/64/16
penryn	sse4_1	penryn	2/64/8:4096/64/16
woodcrest	ssse3	core2	32/64/8:4096/64/16
barcelona	amdsse4a	amdfam10	64/64/2:512/64/16

デフォルト

SPARC および x86 で、-xtarget を指定しないと、-xtarget=generic が想定されます。

拡張

-xtarget オプションは、市販で購入したプラットフォーム上で使用する -xarch、-xchip、-xcache の組み合わせを素早く、簡単に指定するためのマクロです。-xtarget の意味は = のあとに指定した値を展開したものにあります。

例

-xtarget=sun4/15 は -xarch=v8a -xchip=micro -xcache=2/16/1 を意味します。

相互の関連性

-xarch=v9|v9a|v9b オプションで指定する、SPARC V9 アーキテクチャーのコンパイル。-xtarget=ultra や ultra2 の設定は、必要でないか、十分ではありません。-xtarget を指定する場合、-xarch=v9|v9a|v9b オプションは -xtarget よりもあとに表示される必要があります。次に例を示します。

```
-xarch=v9 -xtarget=ultra
```

前述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 よりもあとに指定します。次に例を示します。

```
–xtarget=ultra -xarch=v9
```

警告

別々の手順でコンパイルしてリンクする場合は、コンパイル手順とリンク手順で同じ -xtarget の設定値を使用する必要があります。

A.2.182 -xthreadvar[=*o*]

スレッドローカルな変数の実装を制御するには -xthreadvar を指定します。コンパイラのスレッドローカルな記憶機能を利用するには、このオプションを __thread 宣言指定子と組み合わせて使用します。__thread 指示子を使用してスレッド変数を宣言したあとは、-xthreadvar を指定して動的 (共有) ライブラリの位置に依存しないコード (PIC 以外のコード) でスレッド固有の記憶領域を使用できるようにします。__thread の使用方法の詳細については、67 ページの「4.2 スレッドローカルな記憶装置」を参照してください。

A.2.182.1 値

o には、次のいずれかを指定します。

表 A–50 -xthreadvar の値

値	意味
[no%]dynamic	動的ロード用の変数をコンパイルします [しません]。-xthreadvar=no%dynamic を指定すると、スレッド変数へのアクセスは非常に早くなりますが、動的ライブラリ内のオブジェクトファイルは使用できません。すなわち、実行可能ファイル内のオブジェクトファイルだけが使用可能です。

デフォルト

-xthreadvar を指定しない場合、コンパイラが使用するデフォルトは位置独立コード (PIC) が有効になっているかどうかによって決まります。位置独立コードが有効になっている場合、オプションは -xthreadvar=dynamic に設定されます。位置独立コードが無効になっている場合、オプションは -xthreadvar=no%dynamic に設定されます。

引数を指定しないで -xthreadvar を指定する場合、オプションは -xthreadvar=dynamic に設定されます。

相互の関連性

-mt オプションは、__thread を使用しているファイルのコンパイルおよびリンクを実行するときに指定する必要があります。

警告

動的ライブラリ内に位置に依存するコードがある場合、-xthreadvar を指定する必要があります。

リンカーは、動的ライブラリ内の位置依存コード (非 PIC) スレッド変数と同等のスレッド変数はサポートできません。非 PIC スレッド変数は非常に高速なため、実行可能ファイルのデフォルトとして指定できます。

関連項目

-xcode、 -KPIC、 -Kpic

A.2.183 -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。

A.2.184 -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。

A.2.184.1 値

-xtrigraphs には、次のいずれかを指定します。

表 A–51 -xtrigraphs の値

値	意味
yes	コンパイル単位全体の 3 文字表記の認識を有効にします。
no	コンパイル単位全体の 3 文字表記の認識を無効にします。

デフォルト

コマンド行に -xtrigraphs オプションを指定しなかった場合、コンパイラは -xtrigraphs=yes を使用します。

-xtrigraphs だけを指定すると、コンパイラは -xtrigraphs=yes を使用します。

例

trigraphs_demo.cc という名前のソースファイル例を考えてみましょう。

```
#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。

A.2.185 -xunroll=*n*

可能な場合は、ループを展開します。

このオプションは、コンパイラがループを最適化 (展開) するかどうかを指定します。

A.2.185.1 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。

A.2.186 -xustr={ascii_utf16_ushort|no}

コンパイラにオブジェクトファイル内で UTF-16 文字列に変換させたい文字列リテラルまたは文字リテラルがコードに含まれる場合、を使用します。このオプションが指定されていない場合、コンパイラは 16 ビット文字列リテラルの生成、認識のいずれも行いません。このオプションを使用すれば、U"*ASCII_string*" 文字列リテラルが unsigned short int の配列として認識されます。このような文字列はまだ標準となっていないので、このオプションは、非標準 C++ の認識を可能にします。

すべてのファイルを、このオプションによってコンパイルしなければならないわけではありません。

A.2.186.1 値

ISO10646 UTF--16 文字列リテラルを使用する国際化アプリケーションをサポートする必要がある場合、`-xustr=ascii_utf-16_ushort` を指定します。`-xustr=no` を指定すれば、コンパイラが U"*ASCII_string*" 文字列リテラルまたは文字リテラルを認識しなくなります。このオプションのコマンド行の右端にあるインスタンスは、それまでのインスタンスをすべて上書きします。

`-xustr=ascii_ustf16_ushort` は、U"*ASCII_string*" 文字列リテラルを指定しなくてもかまいません。そのようにしても、エラーとはなりません。

デフォルト

デフォルトは `-xustr=no` です。引数を指定しないで `-xustr` を指定した場合、コンパイラはこの指定を受け付けず、警告を出力します。C または C++ 規格で構文の意味が定義されると、デフォルト値が変わることがあります。

例

次の例では、U を付加した二重引用符で囲んだ文字列リテラルを示します。また、`-xustr` を指定するコマンド行も示します。

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
```

```
const unsigned short *fun() {return foo;}
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

8 ビットの文字列リテラルに U を付加して、unsigned short 型を持つ 16 ビットの UTF-16 文字を形成できます。次に例を示します。

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

A.2.187 -xvector[= *a*]

ベクタライブラリ関数の呼び出しの自動生成や、SIMD (Single Instruction Multiple Data) 命令の生成ができます。このオプションを使用するときは -fround=nearest を指定することによって、デフォルトの丸めモードを使用する必要があります。

-xvector オプションを指定するには、最適化レベルが -x03 かそれ以上に設定されていることが必要です。最適化レベルが指定されていない場合や –x03 よりも低い場合はコンパイルは続行されず、メッセージが表示されます。

a は、次の指定と同じです。

表 A–52 -xvector のフラグ

値	意味
[no%]lib	*(Solaris* のみ)コンパイラは可能な場合はループ内の数学ライブラリへの呼び出しを、同等のベクトル数学ルーチンへの単一の呼び出しに変換します [しません]。大きなループカウントを持つループでは、これによりパフォーマンスが向上します。
[no%]simd	コンパイラはネイティブ x86 SSE SIMD 命令を使用して特定のループのパフォーマンスを向上させます [させません]。ストリーミング拡張機能は、x86 で最適化レベルが 3 かそれ以上に設定されている場合にデフォルトで使用されます。サブオプション no%simd を使用すると、この機能を無効にできます。 コンパイラは、ストリーミング拡張機能がターゲットのアーキテクチャーに存在する場合、つまりターゲットの ISA が SSE2 以上である場合にのみ SIMD を使用します。たとえば、最新のプラットフォームで -xtarget=woodcrest、–xarch=generic64、-xarch=sse2、-xarch=sse3、または -fast を指定して使用できます。ターゲットの ISA にストリーミング拡張機能がない場合、このサブオプションは無効です。
yes	このオプションは、非推奨です。代わりに、-xvector=lib を指定します。
no	このオプションは、非推奨です。代わりに、-xvector=none を指定します。

A.2.187.1 デフォルト

デフォルトは、x86 では -xvector=simd で、SPARC プラットフォームでは -xvector=%none です。サブオプションなしで -xvector を指定すると、コンパイラでは、x86 では -xvector=simd,lib、SPARC (Solaris) では -xvector=lib、および -xvector=simd (Linux) が使用されます。

相互の関連性

コンパイラは、リンク時に libmvec ライブラリを取り込みます。

コンパイルとリンクを別々のコマンドで実行する場合は、リンク時の CC コマンドに必ず -xvector を使用してください。50 ページの「3.3.3 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

A.2.188 -xvis[={yes|no}]

(SPARC) VIS instruction-set Software Developers Kit (VSDK) に定義されているアセンブリ言語のテンプレートを使用する場合は、-xvis=[yes|no] コマンドを 使用します。

VIS 命令セットは、SPARC v9 命令セットの拡張です。UltraSPARC プロセッサは 64 ビットでも、多くの場合、特にマルチメディアアプリケーションではデータサイズが 8 ビットまたは 16 ビットに制限されています。VIS 命令では、1 命令で 4 つの 16 ビットデータを処理できるので、画像、線形代数、信号処理、オーディオ、ビデオ、ネットワーキングなどの新しいメディアを扱うアプリケーションのパフォーマンスが大幅に向上します。

A.2.188.1 デフォルト

デフォルトは -xvis=no です。-xvis と指定すると -xvis=yes と指定した場合と同様の結果が得られます。

A.2.189 - xvpara

OpenMP を使用する場合に正しくない結果をもたらす可能性のある、並列プログラミングに関連する潜在的な問題に関して、警告を発行します。-xopenmp および OpenMP API 指令とともに使用します。

次の状況が検出された場合は、コンパイラは警告を発行します。

- 異なるループ繰り返し間でデータに依存関係がある場合に、MP 指令を使用して並列化されたループ。

- OpenMP データ共有属性節に問題がある場合。たとえば、「shared」と宣言された変数に、OpenMP 並列領域からアクセスするとデータ競合が発生する可能性がある場合や、並列領域の中に値を持つ変数を「private」と宣言し、並列領域よりあとでその変数を使用する場合です。

すべての並列化命令が問題なく処理される場合、警告は表示されません。

次に例を示します。

```
CC -xopenmp -xvpara any.cc
```

注 – Solaris Studio のコンパイラは OpenMP 2.5 API の並列化をサポートします。そのため、MP プラグマ命令は非推奨で、サポートされません。OpenMP API への移植については、『OpenMP API ユーザーズガイド』を参照してください。

A.2.190 -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。

A.2.190.1 関連項目

227 ページの「A.2.16 -errwarn[=*t*]」

A.2.191 -Y*c*,*path*

構成要素 *c* がある場所の新しいパスを指定します。

構成要素の場所を指定する場合、新しいパス名はパス/構成要素名の形式になります。このオプションは ld に渡されます。

A.2.191.1 値

c には次の値のいずれかを指定します。

表 A–53 -Y のフラグ

値	意味
P	cpp のデフォルトのディレクトリを変更します。
0	ccfe のデフォルトのディレクトリを変更します。
a	fbe のデフォルトのディレクトリを変更します。
2	iropt のデフォルトのディレクトリを変更します。

表 A–53 -Y のフラグ　(続き)

値	意味
c (SPARC)	cg のデフォルトのディレクトリを変更します。
O	ipo のデフォルトのディレクトリを変更します。
k	CClink のデフォルトのディレクトリを変更します。
l	ld のデフォルトのディレクトリを変更します。
f	c++filt のデフォルトのディレクトリを変更します。
m	mcs のデフォルトのディレクトリを変更します。
u *(x86)*	ube のデフォルトのディレクトリを変更します。
h *(x86)*	ir2hf のデフォルトのディレクトリを変更します。
A	すべてのコンパイラ構成要素を検索するときのディレクトリを指定します。構成要素がパスにない場合は、コンパイラがインストールされているディレクトリに戻って検索が行われます。
P	デフォルトのライブラリ検索パスにパスを追加します。デフォルトのライブラリ検索パスの前にこのパスが調べられます。
S	起動用のオブジェクトファイルのデフォルトのディレクトリを変更します。

相互の関連性

コマンド行に複数の -Y オプションを配置できます。2 つ以上の -Y オプションが 1 つの項目に適用されている場合には、最後に指定されたものが有効です。

関連項目

リンカーとライブラリ

A.2.192 -z[ ]*arg*

リンクエディタのオプション。詳細は、ld(1) のマニュアルページと Solaris 関連のマニュアル『リンカーとライブラリ』を参照してください。

◆ ◆ ◆ 付録 B

プラグマ

この付録では、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++ 標準の一部ですが、書式、内容、および意味はコンパイラごとに異なります。プラグマは C++ 標準には定義されていません。

注 – したがってプラグマに依存するコードには移植性はありません。プラグマに依存するコードは移植性がありません。

B.1 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。

B.1.1 プラグマの引数としての多重定義関数

ここで示すいくつかのプラグマは、引数として関数名をとります。その関数が多重定義されている場合、プラグマは、その引数として、その直前の関数宣言を使用します。次の例を考えてみましょう。

```
int bar(int);
int foo(int);
```

```
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

この例の foo は、プラグマの直前の foo の宣言である foo(double) を意味し、bar は、単に宣言されている bar である bar(int) を意味します。ここで、foo が再び多重定義されている次の例を考えてみます。

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

この例の bar は、単に宣言されている bar である bar(int) を意味します。しかし、プラグマは、どのバージョンの foo を使用すべきか分かりません。この問題を解決するには、プラグマが使用すべき foo の定義の直後にプログラムを置く必要があります。

次のプラグマは、この節で説明した方法で選択を行います。

- does_not_read_global_data
- does_not_return
- does_not_write_global_data
- no_side_effect
- opt
- rarely_called
- returns_new_memory

B.2 プラグマの詳細

この節では、C++ コンパイラにより認識されるプラグマキーワードについて説明します。

B.2.1 #pragma align

#pragma align *integer*(*variable* [,*variable*...])

align を使用すると、指定したすべての変数のメモリー境界を *integer* 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いけません。
- *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。
- 指定された境界がデフォルトより小さい場合は、デフォルトが優先します。

- この #pragma 行は、指定した変数の宣言より前になければいけません。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;
}
```

B.2.2 #pragma does_not_read_global_data

#pragma does_not_read_global_data(*funcname* [, *funcname*])

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを読み込まないことをコンパイラに宣言します。この表明により、そうしたルーチンへの呼び出し前後のコードをさらに最適化することができます。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとに限定されます。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.3 #pragma does_not_return

#pragma does_not_return(*funcname* [, *funcname*])

指定した関数への呼び出しが復帰しないことをコンパイラに表明します。この表明により、コンパイラは、指定された関数への呼び出しが戻らないと仮定して最適化を行うことができます。たとえば、呼び出し側で終了したライフ回数を登録すると、より多くの最適化が可能となります。

指定した関数が復帰した場合は、プログラムの動作は未定義になります。

次の例のとおり、このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとに限定されます。

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.4 #pragma does_not_write_global_data

`#pragma does_not_write_global_data(`*funcname* `[, `*funcname*`])`

このプラグマは、指定したルーチンが直接的にも間接的にも大域データを書き込まないことをコンパイラに宣言します。この表明により、そうしたルーチンへの呼び出し前後のコードをさらに最適化することができます。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

このプラグマを使用できるのは、指定した関数のプロトタイプを宣言したあとに限定されます。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.5 #pragma dumpmacro s

`#pragma dumpmacros (`*value*`[,`*value*`...])`

マクロがプログラム内でどのように動作しているかを調べたいときに、このプラグマを使用します。このプラグマは、定義済みマクロ、解除済みマクロ、実際の使用状況といった情報を提供します。マクロの処理順序に従って、標準エラー (`stderr`) に出力します。`dumpmacros` プラグマは、ファイルが終わるまで、または `#pragma end_dumpmacro` に到達するまで、有効です。377 ページの「B.2.6 #pragma end_dumpmacros」を参照してください。 *value* の代わりに次の引数を使用できます。

値	意味
defs	すべての定義済みマクロを出力します
undefs	すべての解除済みマクロを出力します

値	意味
use	使用されているマクロの情報を出力します
loc	defs、undefs、use の位置 (パス名と行番号) も出力します
conds	条件付き指令で使用したマクロの使用情報を出力します
sys	システムヘッダーファイルのマクロについて、すべての定義済みマクロ、解除済みマクロ、使用状況も出力します

注 – サブオプション loc、conds、sys は、オプション defs、undefs、use の修飾子です。loc、conds、sys は、単独では効果はありません。たとえば #pragma dumpmacros=loc,conds,sys には、何も効果はありません。

dumpmacros プラグマとコマンド行オプションの効果は同じですが、プラグマがコマンド行オプションを上書きします。304 ページの「A.2.122 -xdumpmacros[=*value*[,*value*...]]」を参照してください。

dumpmacros プラグマは入れ子にならないので、次のコードでは #pragma end_dumpmacros が処理されるとマクロ情報の出力が停止します。

```
#pragma dumpmacros (defs, undefs)
#pragma dumpmacros (defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros プラグマの効果は累積的です。次のものは、

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

次と同じ効果を持ちます。

```
#pragma dumpmacros(defs, undefs, loc)
```

オプション #pragma dumpmacros=use,no%loc を使用した場合、使用したマクロそれぞれの名前が一度だけ出力されます。オプション #pragma dumpmacros=use,loc を使用した場合、マクロを使用するたびに位置とマクロ名が出力されます。

B.2.6 #pragma end_dumpmacros

```
#pragma end_dumpmacros
```

このプラグマは、dumpmacrospragma が終わったことを通知し、マクロ情報の出力を停止します。dumpmacros プラグマ終了時に end_dumpmacros プラグマを使用しなかった場合、dumpmacros プラグマはファイルが終わるまで出力を生成し続けます。

B.2.7 #pragma error_messages

#pragma error_messages (on|off|default, *tag*... *tag*)

このエラーメッセージプラグマは、ソースプログラムの中から、コンパイラが発行するメッセージを制御可能にします。プラグマは警告メッセージにのみ効果があります。-w コマンド行オプションは、すべての警告メッセージを無効にすることでこのプラグマを上書きします。

- #pragma error_messages (on, *tag*... *tag*)

  on オプションは、先行する #pragma error_messages オプション (off オプションなど) をその時点で無効にして、-erroff オプションも無効にします。

- #pragma error_messages (off, *tag*... *tag*)

  off オプションは、コンパイラプログラムが指定トークンから始まる特定のメッセージを発行することを禁止します。この特定のエラーメッセージに対するプラグマの指定は、別の #pragma error_messages によって無効にされるか、コンパイルが終了するまで有効です。

- #pragma error_messages (default, *tag*... *tag*)

  default オプションは、指定タグについて、先行する #pragma error_messages 指令を無効にします。

B.2.8 #pragma fini

#pragma fini (*identifier*[,*identifier*...])

fini を使用すると、*identifier* 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あとに実行されます。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。

このような関数は #pragma fini 指令の中に登場するたびに、1 回呼び出されます。

B.2.9 #pragma hdrstop

hdrstop プラグマをソースファイルヘッダーに埋め込むと、活性文字列の終わりが指示されます。たとえば次のファイルがあるとします。

```
example% cat a.cc
#include "a.h"
#include "b.h"
```

```
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

活性文字列は c.h で終わるので、各ファイルの c.h の後に #pragma hdrstop を挿入します。

#pragma hdrstop を挿入できる場所は、CC コマンドで指定したソースファイルの活性文字列の終わりだけです。#pragma hdrstop をインクルードファイル内に指定しないでください。

338 ページの「A.2.162 -xpch=*v*」および 341 ページの「A.2.163 -xpchstop=*file*」 を参照してください。

B.2.10 #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できます。

B.2.11 #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あとに実行されます。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。

このような関数は #pragma init 指令の中に登場するたびに、1 回呼び出されます。

B.2.12 #pragma must_have_frame

#pragma must_have_frame(*funcname* [,*funcname*])

このプラグマは、(System V ABI で定義されているとおり) 完全なスタックフレームを必ず持つように、指定した関数リストをコンパイルすることを要求します。このプラグマで関数を列挙する前に、関数のプロトタイプを宣言する必要があります。

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

このプラグマを使用できるのは、指定した関数のプロトタイプの宣言後のみに限定されます。プラグマは関数の最後より先に記述する必要があります

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.13 #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、次の処理も行なってはいけません。

- 呼び出し時点で呼び出し側が認識できるプログラム状態の一部に、読み出しまたは書き込みのためにアクセスすることはありません。
- 入出力を実行しません。
- 呼び出し時点で認識できるプログラム状態のどの部分も変更しません。

コンパイラは、この情報を最適化に使用します。

関数に副作用があると、この関数を呼び出すプログラムの実行結果は未定義になります。

name 引数で、現在の翻訳単位に含まれている関数の名前を指定します。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければいけません。プラグマは、関数定義の前に位置していなければいけません。

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.14 #pragma opt

`#pragma opt` *level* (*funcname[, funcname]*)

funcname には、現在の翻訳単位内で定義されている関数の名前を指定します。*level* の値は、指定した関数に対する最適化レベルです。0、1、2、3、4、5 いずれかの最適化レベルを割り当てることができます。*level* を 0 に設定すると、最適化を無効にできます。関数は、プラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。プラグマは、最適化する関数の定義を処理する必要があります。

プラグマ内に指定される関数の最適化レベルは、`-xmaxopt` の値に下げられます。`-xmaxopt=off` の場合、プラグマは無視されます。

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.15 #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 を指定する場合は、0 または 2 の累乗にする必要があります。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、指令のあと (および後続の `pack` 指令の前) に定義されているすべての構造体のメンバーを 2 バイト境界を超えないように揃えます。

```
#pragma pack(2)
```

n が 0 であるか省略された場合、メンバー整列は自然境界整列の値に戻ります。

n の値がプラットフォームのもっとも厳密な境界整列と同じかそれ以上の場合には、自然境界整列になります。次の表に、各プラットフォームのもっとも厳密な境界整列を示します。

表 B–1 プラットフォームのもっとも厳密な境界整列

プラットフォーム	もっとも厳密な境界整列
x86	4
SPARC 一般、V8、V8a、V8plus、V8plusa、V8plusb	8
SPARC V9、V9a、V9b	16

pack 指令は、次の pack 指令までに存在するすべての構造体定義に適用されます。別々の翻訳単位にある同じ構造体に対して異なる境界整列が指定されると、プログラムは予測できない状態で異常終了する場合があります。特に、コンパイル済みライブラリのインタフェースを定義するヘッダーをインクルードする場合は、その前に pack を使用しないでください。プログラムコード内では、pack 指令は境界整列を指定する構造体の直前に置き、#pragma pack() は構造体の直後に置くことをお勧めします。

SPARC プラットフォーム上で #pragma pack を使用して、型のデフォルトの境界整列よりも密に配置するには、アプリケーションのコンパイルとリンクの両方で -misalign オプションを指定する必要があります。次の表に、整数データ型のメモリーサイズとデフォルトの境界整列を示します。

表 B–2 メモリーサイズとデフォルトの境界整列 (単位はバイト数)

種類	SPARC V8 サイズ、境界整列	SPARC V9 サイズ、境界整列	x86 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
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

B.2.16 #pragma rarely_called

#pragms rarely_called(*funcname[, funcname]*)

このプラグマは、指定の関数がほとんど呼び出されないことをコンパイラに示唆します。このヒントにより、コンパイラは、プロファイル収集段階に負担をかけることなく、ルーチンの呼び出し元でプロファイルフィードバック方式の最適化を行うことができます。このプラグマはヒントの提示ですので、コンパイラは、このプラグマに基づく最適化を行わないこともあります。

`#pragma rarely_called` プリプロセッサ指令を使用できるのは、指定の関数のプロトタイプが宣言されたあとだけです。次は、`#pragma rarely_called` の例です。

```
extern void error (char *message);
#pragma rarely_called(error)
```

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.17 #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、そのポインタがほかのポインタの別名として使用されないことをコンパイラに宣言します。この情報により、オプティマイザはポインタ値をより正確に追跡し、メモリー位置を明確化することができます。この結果、スケジューリングとパイプライン化が改善されます。

このプラグマの宣言が実際には誤っている場合は、該当する関数を呼び出したプログラムの実行結果は保証されません。

name 引数で、現在の翻訳単位に含まれている関数の名前を指定します。プラグマは関数と同じスコープ内になければならず、また、関数宣言後に位置していなければいけません。プラグマは、関数定義の前に位置していなければいけません。

プラグマがその引数として多重定義関数を処理する方法の詳細は、373 ページの「B.1.1 プラグマの引数としての多重定義関数」を参照してください。

B.2.18 #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を指定できます。たとえば、`setjmp()` の直後の文は、ほかのどんなルーチンを呼び出してもそこから返ってくることができます。これは、`longjmp()` を呼び出すことによって行います。

このようなルーチンを使用すると標準のフローグラフ解析ができないため、呼び出す側のルーチンを最適化すると安全性が確保できません。このような場合に #pragma unknown_control_flow を使用すると安全な最適化が行えます。

関数名が多重定義されている場合、最後に宣言された関数が選ばれます。

B.2.19 #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。

- 文字列書式。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いけません。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。
- 識別子書式。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いけません。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。

B.2.19.1 #pragma weak *name*

#pragma weak *name* という書式の指令は、*name* を弱い (weak) シンボルに定義します。*name* のシンボル定義が見つからなくても、リンカーはエラーメッセージを生成しません。また、弱いシンボルの定義を複数見つけた場合でも、リンカーはエラーメッセージを生成しません。リンカーは単に最初に検出した定義を使用します。

プラグマ name の強い定義が存在しない場合、リンカーはシンボルの値を 0 にします。

次の指令は、ping を弱いシンボルに定義しています。ping という名前のシンボルの定義が見つからない場合でも、リンカーはエラーメッセージを生成しません。

```
#pragma weak ping
```

#pragma weak *name1* = *name2*

#pragma weak *name1* = *name2* という書式の指令は、シンボル *name1* を *name2* への弱い参照として定義します。*name1* がどこにも定義されていない場合、*name1* の値は *name2* の値になります。*name1* が別の場所で定義されている場合、リンカーはその定義を使用し、*name2* への弱い参照は無視します。次の指令では、bar がプログラムのどこかで定義されている場合、リンカーはすべての参照先を bar に設定します。そうでない場合、リンカーは bar への参照を foo にリンクします。

```
#pragma weak bar = foo
```

識別子書式では、*name2* は現在のコンパイル単位内で宣言および定義しなければいけません。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 つしか存在してはいけません。多重定義関数に識別子書式の #pragma weak を使おうとすると、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        // error, ambiguous function name
```

このエラーを回避するには、文字列書式を使用します。例を次に示します。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

詳細は、Solaris の『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。
ELF ファイル	ELF は Executable and Linking Format の略語で、コンパイラによって生成されるファイル。
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。
stack	あと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。
switch	「コンパイラオプション」を参照。
type	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 `integer` と `float` です。ほかのすべての型は、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。
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。
関数の多相性	「関数の多重定義」参照。
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、および非公開があります。
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。同義語: フラグ、スイッチ。
最適化	コンパイラが生成するオブジェクトコードの効率を良くする処理のこと。
サブルーチン	関数のこと。Fortran では、値を返さない関数を指します。
左辺値	変数のデータ値が格納されているメモリーの場所を表す式。あるいは、代入演算子の左辺にある変数のインスタンス。

事後束縛	「動的束縛」を参照。
事前束縛	「静的束縛」を参照。
実行時型識別機構 **(RTTI)**	プログラムが実行時にオブジェクトの型を識別できるようにする標準的な方法を提供する仕組み。
実行時束縛	「動的束縛」を参照。
シンボル	何らかのプログラムエントリを示す名前やラベル。
シンボルテーブル	プログラムのコンパイルで検出されたすべての識別子と、それらのプログラム中の位置と属性からなるリスト。コンパイラは、このテーブルを使って識別子の使い方を判断します。
スコープ	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。
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。
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、ほかのクラスが抽象クラスを拡張し、その抽象メソッドを実装することで具体化されることを目的として、定義されています。
抽象メソッド	実装を持たないメソッド。
データ型	文字、整数、浮動小数点数などを表現するための仕組み。変数に割り当てられる記憶域とその変数に対してどのような演算が実行可能かは、この型によって決まります。
データメンバー	クラスの要素であるデータ。関数や型定義と区別してこのように呼ばれます。
デストラクタ	クラスオブジェクトを破棄したり、演算子 `delete` をクラスポインタに適用したときにコンパイラによって自動的に呼び出される特別なクラスメンバー関数。デストラクタの名前は、それが属するクラスの名前と同じで、かつ、名前の前にチルド (~) が必要です。「コンストラクタ」を参照。
テンプレートオプションファイル	テンプレートのコンパイル用オプションやソースの位置などの情報が含まれている、ユーザーが用意するファイル。テンプレートオプションファイルの使用は推奨されていないため、使用すべきではありません。
テンプレートデータベース	プログラムが必要とするテンプレートの処理とインスタンス化に必要なすべての構成ファイルを含むディレクトリ。

テンプレートの特殊化	デフォルトのインスタンス化では型を適切に処理できないときに、このデフォルトを置き換える、クラステンプレートメンバー関数の特殊インスタンス。
動的キャスト	ポインタや参照の型を、宣言されたものから、それが参照する動的な型と矛盾しない任意の型に安全に変換するための方法。
動的束縛	関数呼び出しと関数本体を実行時に結び付けること。これは、仮想関数に対してのみ行われます。事後束縛または実行時束縛とも呼ばれます。
動的な型	ポインタや参照でアクセスするオブジェクトの実際の型。この型は、宣言された型と異なることがあります。
トラップ	ほか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、プログラム制御がほか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。「名前修飾」とも呼びます。
バイナリ互換	あるリリースのコンパイラでコンパイルしたオブジェクトファイルを別のリリースのコンパイラを使用してリンクできること。
配列	同じデータ型の値をメモリーに連続して格納するデータ構造。各値にアクセスするには、配列内のそれぞれの値の位置を指定します。
派生クラス	「継承」を参照。
符号化する	「名前の符号化」を参照。
フラグ	「コンパイラオプション」を参照。
プラグマ	コンパイラに特定の処置を指示するコンパイラのプリプロセッサ命令、または特別な注釈。
べき等	ヘッダーファイルの属性。ヘッダーファイルを 1 つの翻訳単位に何回インクルードしても、一度インクルードした場合と同じ効果を持つこと。
変数	識別子で命名されているデータ項目。各変数は `int` や `void` などの型とスコープを持っています。「クラス変数」、「インスタンス変数」、「局所変数」も参照。
マルチスレッド	シングルまたはマルチプロセッサシステムで並列アプリケーションを開発・実行するためのソフトウェア技術。
メソッド	一部のオブジェクト指向言語でメンバー関数の代わりに使用される用語。
メンバー関数	クラスの要素である関数。データ定義や型定義と区別してこのように呼ばれます。

リンカー	オブジェクトコードとライブラリを結び付けて、完全な実行可能プログラムを作成するツール。
例外	プログラムの通常の流れの中で起こる、プログラムの継続を妨げるエラー。たとえば、メモリーの不足やゼロ除算などを指します。
例外処理	エラーの捕捉と防止を行うためのエラー回復処理。具体的には、プログラムの実行中にエラーが検出されると、あらかじめ登録されている例外ハンドラにプログラムの制御が戻り、エラーを含むコードは実行されなくなることを指します。
例外ハンドラ	エラーを処理するために作成されたコード。ハンドラは、対象とする例外が起こると自動的に呼び出されます。
ロケール	地理的な領域と言語のどちらか、あるいはその両方に固有な一連の規約。日付、時刻、通貨単位などの形式。

索引

数字・記号

, 72
+d, コンパイラオプション, 221
+e(0|1), コンパイラオプション, 225
#error, 42
\>\> 抽出演算子
  complex, 204
  iostream, 180
+p, コンパイラオプション, 267
#pragma align, 374
#pragma does_not_read_global_data, 375
#pragma does_not_return, 375
#pragma does_not_write_global_data, 376
#pragma dumpmacros, 376–377
#pragma end_dumpmacros, 377
#pragma error_messages, 378
#pragma fini, 378
#pragma ident, 379
#pragma init, 379
#pragma must_have_frame, 380
#pragma no_side_effect, 380
#pragma opt, 381
#pragma pack, 381
#pragma rarely_called, 383
#pragma returns_new_memory, 383
#pragma unknown_control_flow, 383
#pragma weak, 384
#pragma キーワード, 374–385
+w, コンパイラオプション, 281
+w2, コンパイラオプション, 282
#warning, 42
+w、コンパ　ラオプション, 97
-386, コンパイラオプション, 217
3 文字表記シーケンス、認識する, 365
-486, コンパイラオプション, 217

A

.a, ファイル名接尾辞, 36
Apache C++ 標準ライブラリ, 259
ATS: 自動チューニングシステム, 342
.a、ファイル名接尾辞, 207

B

-B結合, コンパイラオプション, 110
-B*binding*, コンパイラオプション, 217
bool 型とリテラル、許可する, 232

C

.c++, ファイル名接尾辞, 36
C++ 標準ライブラリ, 142
  RogueWave Version, 157
  置き換え, 151–155
  構成要素, 157
  マニュアルページ, 144, 159–172
C++ マニュアルページ, アクセス, 144
c_exception, complex class, 203
.c, ファイル名接尾辞, 36
.C, ファイル名接尾辞, 36
-c, コンパイラオプション, 38, 219

C99 サポート，317
.cc, ファイル名接尾辞，36
CCadmin コマンド，97
CCFLAGS, 環境変数，45–46
cc コンパイラオプション -xaddr32，282
cerr 標準ストリーム，130
cerr 標準ストリーム，175
char, 符号性，296
char* 抽出子，181–182
char の有符号性，296
char の有符号性の保護，296
cin 標準ストリーム，130, 175
clog 標準ストリーム，130, 175
-compat
  C++ ライブラリのリンクのモード，148
  コンパイラオプション，219
  デフォルトでリンクされるライブラリへの影響，144
  ライブラリを使用できるモード，141
complex
  constructors，200
  エラー処理，203
  演算子，201
  効率，206
  互換モード，199
  混合算術演算，205
  三角関数，203
  数学関数，202
  入力/出力，204
  標準モードと libCstd，199
  ヘッダーファイル，200
  マニュアルページ，206
  ライブラリ，142, 147–148, 199–200
  ライブラリ, リンク，200
complex_error
  definition，203
  メッセージ，202
cout, 標準ストリーム，130, 175
__cplusplus, 事前定義マクロ，75, 219
.cpp, ファイル名接尾辞，36
.cxx, ファイル名接尾辞，36
C インタフェース
  C++ 実行時ライブラリに依存しないようにする，211
C インタフェース (続き)
  ライブラリの作成，211
C 標準ヘッダーファイル, 置き換え，155

D

-D_REENTRANT，124
-D, コンパイラオプション，48, 222
-d, コンパイラオプション，222–223
-dalign, コンパイラオプション，223
-DDEBUG，104
dec, iostream マニピュレータ，188
delete 配列形式, 認識する，234
dlclose(), 関数呼び出し，209
dlopen(), 関数呼び出し，208
double, complex 型の値，200
-dryrun, コンパイラオプション，40, 224
dwarf デバッガデータ形式，302
.d ファイル拡張子，325

E

-E, コンパイラオプション，224
EDOM, errno の設定，204
elfdump，301
endl, iostream マニピュレータ，188
ends, iostream マニピュレータ，188
enum
  スコープ修飾子、として名前を使用，69–70
  前方宣言，68–69
  不完全な、使用，69
er_src ユーティリティー，293
ERANGE, errno の設定，204
errno
  definition，203
  -fast との相互関係，230
  の値の保持，230
-erroff, コンパイラオプション，226
error 関数，179
-errtags, コンパイラオプション，227
-errwarn, コンパイラオプション，227
explicit キーワード, 認識する，234
export キーワード, 認識する，232

F

-fast, コンパイラオプション，228–231
-features, コンパイラオプショ，118
-features, コンパイラオプション，65, 108, 231
files
　「source files」も参照
-filt, コンパイラオプション，235
-flags, コンパイラオプション，237
float 型挿入子、iostream 出力，178
flush, iostream マニピュレータ，180
flush, iostream マニピュレータ，188
-fnonstd, コンパイラオプション，238
-fns, コンパイラオプション，238–240
Fortran 実行時ライブラリ, リンク，317
-fprecision=*p*, コンパイラオプション，240
-fround=*r*, コンパイラオプション，241–242
-fsimple=*n*, コンパイラオプション，242
-fstore, コンパイラオプション，243
fstream.h
　iostream ヘッダーファイル，177
　使用，184
fstream, 定義，176
fstream、定義，196
-ftrap, コンパイラオプション，244
__func__、識別子，72

G

-G
　オプションの説明，245
　動的ライブラリコマンド，209
-g
　オプションの説明，246
　によるテンプレートのコンパイル，104
get, char 抽出子，182
get ポインタ, streambuf，192
__global，66
-g0 オプションの説明，248

H

-H, コンパイラオプション，248
-h, コンパイラオプション，248–249
-help, コンパイラオプション，249
hex, iostream マニピュレータ，188
__hidden，66

I

I/O ライブラリ，175
.i, ファイル名接尾辞，36
-I-, コンパイラオプション，250–252
-I, コンパイラオプション，105, 249–250
-i, コンパイラオプション，252
ifstream, 定義，176
.il, ファイル名接尾辞，36
-include, コンパイラオプション，252
include ディレクトリ、テンプレート定義ファイル，105
include ファイル, 検索順序，249, 250–252
-inline,「-xinline」参照，253
-instances=*a*, コンパイラオプション，253
-instances=*a*、コンパイラオプション，99
-instlib, コンパイラオプション，254
iomanip.h, iostream ヘッダーファイル，177, 189
iostream
　stdio，184, 192
　エラー処理，183–184
iostream, エラービット，180
iostream
　機能の拡張, マルチスレッドでの安全性における注意事項，133
　構造，176–177
　互換モード，175
　コピー，187
　コンストラクタ，176
　作成，184–187
　事前定義，175–176
　従来の iostream，143, 146, 261
　出力エラー，179–180
　使用，177–184
　新旧の形式が混在する，261
　シングルスレッドアプリケーション，124
　ストリームの代入，187
　定義，196
　入力，180–181
　標準 iostream，143, 146, 261

iostream (続き)
標準モード, 175, 177, 261
フォーマット, 187
フラッシュ, 180
ヘッダーファイル, 177
マニピュレータ, 188–191
マニュアルページ, 175, 193–195
マルチスレッドで使用しても安全な新しいインタフェース関数, 128
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, 128–130
マルチスレッドで使用しても安全な再入可能な関数, 124
マルチスレッドで使用しても安全な制約, 124–126
マルチスレッドにおける新しいクラス階層, 128
用語, 195–197
ライブラリ, 142, 146–147, 148
～への出力, 178–180
iostream.h, iostream ヘッダーファイル, 130
iostream.h、iostream ヘッダーファイル, 177
ISO10646 UTF-16 文字列リテラル, 367
ISO C++ 標準
準拠, 30
単一定義規則, 93, 104
istream クラス、定義, 176
istrstream クラス, 定義, 176

K

-keeptmp, コンパイラオプション, 256
-Kpic, コンパイラオプション, 209, 256
-KPIC, コンパイラオプション, 209, 255

L

-L, コンパイラオプション, 144, 256
-l, コンパイラオプション, 48, 141, 144, 256
LD_LIBRARY_PATH 環境変数, 208
libC
MT 環境、における使用, 121
互換モード, 175, 177
libC (続き)
マルチスレッドで使用しても安全な新しいクラス, 128
マルチスレッドでの安全性のコンパイルとリンク, 124
ライブラリ, 142
libCrun ライブラリ, 117, 118, 142, 144
libCstd ライブラリ,「C++ 標準ライブラリ」を参照
libcsunimath, ライブラリ, 142
libc ライブラリ, 141
libdemangle ライブラリ, 142
libgc ライブラリ, 142
libiostream,「iostream」を参照
libm
インラインテンプレート, 320
オプティマイズされたバージョン, 321
ライブラリ, 141
-libmieee, コンパイラオプション, 257
-libmil, コンパイラオプション, 257
-library, コンパイラオプション, 144, 149, 257–262
librwtool,「Tools.h++」を参照, 143
libthread ライブラリ, 141
limit, コマンド, 44
linking, iostream library, 147
-lthread コンパイラオプション
-xnolib による抑止, 150
代わりに -mt を使用, 117, 124

M

math.h, complex ヘッダーファイル, 206
mbarrier.h, 136–137
-mc, コンパイラオプション, 263
-migration, コンパイラオプション, 263
-misalign, コンパイラオプション, 263–264
-mr, コンパイラオプション, 264
-mt オプション, および libthread, 124
-mt コンパイラオプション
オプションの説明, 264–265
ライブラリのリンク, 141
mutable キーワード, 認識, 232

N

namespace キーワード, 認識する，234
-native, コンパイラオプション，265
nestedacess キーワード，232
new 配列形式, 認識する，234
-noex, コンパイラオプション，265
noex, コンパイラオプション，118
-nofstore, コンパイラオプション，265
-nolib, コンパイラオプション，145, 265
-nolibmil, コンパイラオプション，266
-noqueue, コンパイラオプション，266
-norunpath, コンパイラオプション，146, 266
! NOT 演算子, iostream，179, 183

O

-O, コンパイラオプション，266
-o, コンパイラオプション，267
oct, iostream マニピュレータ，188
ofstream クラス，184
ostream クラス、定義，176
ostrstream クラス, 定義，176
output, cout，178
overflow 関数, streambuf，133
.o ファイル
　オプション接尾辞，36
　残す，38

P

-P, コンパイラオプション，268
PEC: 移植可能な実行可能コード，342
Pentium，363
-pentium, コンパイラオプション，268
-pg, コンパイラオプション，268
-PIC, コンパイラオプション，268
-pic, コンパイラオプション，268
POSIX スレッド，264–265
private, オブジェクトスレッド，131
-pta, コンパイラオプション，269
ptclean コマンド，97
pthread_cancel() 関数，118
-pti, コンパイラオプション，105, 269
-pto, コンパイラオプション，269
-ptr, コンパイラオプション，269
-ptv, コンパイラオプション，269
put ポインタ、streambuf，192

Q

-Qoption, コンパイラオプション，270
-qoption, コンパイラオプション，271
-qp, コンパイラオプション，271
-Qproduce, コンパイラオプション，271
-qproduce, コンパイラオプション，271

R

-R, コンパイラオプション，146, 272
-readme, コンパイラオプション，272
readme ファイル，30
reinterpret_cast 演算子，285
resetiosflags, iostream マニピュレータ，189
RogueWave
　C++ 標準ライブラリ，157
　「Tools.h++」も参照，143
rtti キーワード、認識する，234
rvalueref キーワード，232

S

.s, ファイル名接尾辞，36
.S, ファイル名接尾辞，36
-S, コンパイラオプション，272
-s, コンパイラオプション，272–273
-sb, コンパイラオプション，273
-sbfast, コンパイラオプション，273
sbufpub, マニュアルページ，185
set_terminate() 関数，118
set_unexpected() 関数，118
setbase, iostream マニピュレータ，188
setfill, iostream マニピュレータ，189
setiosflags, iostream マニピュレータ，189
setprecision, iostream マニピュレータ，189
setw, iostream マニピュレータ，188

skip フラグ、iostream, 183
.so.*n*, ファイル名接尾辞, 36
.so, ファイル名接尾辞, 36, 207
Solaris オペレーティング環境ライブラリ, 141
Solaris スレッド, 264–265
stabs デバッガデータ形式, 302
『Standard C++ Class Library Reference』, 158
-staticlib, コンパイラオプション, 145, 149, 273
__STDC__, 事前定義マクロ, 75
stdcxx4 キーワード, 259
stdio
  iostream との, 184
  stdiobuf マニュアルページ, 192
stdiostream.h, iostream ヘッダーファイル, 177
STLport, 173
STL (標準テンプレートライブラリ), 構成要素, 157
stream.h, iostream ヘッダーファイル, 177
stream_locker
  マニュアルページ, 133
  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, 127
streambuf
  get ポインタ, 192
  put ポインタ, 192
  新しい関数, 128
  キュー形式とファイル形式, 192
  公開仮想関数, 133
  使用, 193
  定義, 196
  で定義された, 192
  マニュアルページ, 193
  ロック, 123
streampos, 186
strstream.h, iostream ヘッダーファイル, 177
strstream, 定義, 176
strstream、定義, 197
struct、名前のない宣言, 70
__SUNPRO_CC_COMPAT=(4|5), 事前定義マクロ, 219
.SUNWCCh ファイル名接尾辞, 153
SunWS_cache, 103
swap -s, コマンド, 43
__symbolic, 66
-sync_stdio, コンパイラオプション, 275

T

tcov, -xprofile, 352
-temp=*dir*, コンパイラオプション, 276
-template, コンパイラオプション, 105, 276
-template、コンパイラオプション, 98
terminate() 関数, 118
thr_keycreate, マニュアルページ), 131
__thread, 67
-time, コンパイラオプション, 278
Tools.h++
  コンパイラオプション, 149
  従来の iostream と標準 iostream, 143
  デバッグ ライブラリ, 142
  ドキュメント, 143
  標準モードと互換モード, 143
traceback, 278–279
-traceback, コンパイラオプション, 278–279

U

U"..." 形式の文字列リテラル, 367
-U, コンパイラオプション, 48, 279
ulimit, コマンド, 44
unexpected() 関数, 118
-unroll=*n*, コンパイラオプション, 279

V

-V, コンパイラオプション, 279
-v, コンパイラオプション, 40, 280
__VA_ARGS__ 識別子, 41–42
values, flush, 180
-vdelx, コンパイラオプション, 280
-verbose, コンパイラオプション, 97, 280
VIS Software Developers Kit, 369

W

-w, コンパイラオプション, 282
ws, iostream マニピュレータ, 183, 188

X

-xalias_level, コンパイラオプション，283
-xannotate, コンパイラオプション，285–286
-xar, コンパイラオプション，208, 286
-xarch=*isa*, コンパイラオプション，287
-xar、コンパイラオプション，100
-xautopar, コンパイラオプション，292
-xbinopt, コンパイラオプション，292
-xbinoptコンパイラオプション，292
-xbuiltin, コンパイラオプション，293
-xcache=*c*, コンパイラオプション，294–295
-xcg, コンパイラオプション，219, 295
-xcg89, コンパイラオプション，295
-xchar, コンパイラオプション，296
-xcheck, コンパイラオプション，297
-xchip=*c*, コンパイラオプション，298
-xcode=*a*, コンパイラオプション，300–302
-xdebugformatコンパイラオプション，302
-xdepend, コンパイラオプション，303
-xdumpmacros, コンパイラオプション，304
-xe, コンパイラオプション，307
-xF, コンパイラオプション，307–308
-xhelp=flags, コンパイラオプション，308
-xhelp=readme, コンパイラオプション，308
-xhreadvar, コンパイラオプション，364
-xhwcprofコンパイラオプション，309
-xia, コンパイラオプション，309
-xinline, コンパイラオプション，310
-xipo_archiveコンパイラオプション，315
-xipo, コンパイラオプション，313
-xjobs, コンパイラオプション，316
-xkeepframe, コンパイラオプション，317
-xlang, コンパイラオプション，317
-xldscope, コンパイラオプション，65, 319
-xlibmieee, コンパイラオプション，320
-xlibmil, コンパイラオプション，320–321
-xlibmopt, コンパイラオプション，321
-xlic_lib, コンパイラオプション，321
-xlicinfo, コンパイラオプション，321
-xlinkopt, コンパイラオプション，322
-xloopinfo, コンパイラオプション，323
-Xm, コンパイラオプション，282
-xM, コンパイラオプション，323–324
-xM1, コンパイラオプション，324
-xmaxopt, コンパイラオプション，326
-xmaxopt コンパイラオプション，326
-xMD, コンパイラオプション，325
-xmemalign, コンパイラオプション，326
-xMerge, コンパイラオプション，325–326
-xMF, コンパイラオプション，325
-xMMD, コンパイラオプション，325
-xmodel, コンパイラオプション，328
-xnolib, コンパイラオプション，145, 149, 329
-xnolibmil, コンパイラオプション，330
-xnolibmopt, コンパイラオプション，330–331
-xO*level*,，267
-xO*level*, コンパイラオプション，331–334
-xopenmp, コンパイラオプション，334
-xpagesize_heap, コンパイラオプション，337
-xpagesize_stack, コンパイラオプション，337
-xpagesize, コンパイラオプション，336
-xpec, コンパイラオプション，342
-xpg, コンパイラオプション，342–343
-xport64, コンパイラオプション，343
-xprefetch_auto_type, コンパイラオプション，349
-xprefetch_level, コンパイラオプション，349
-xprefetch, コンパイラオプション，346
-xprofile_ircache, コンパイラオプション，353
-xprofile_pathmap, コンパイラオプション，354
-xreduction, コンパイラオプション，354
-xregs, コンパイラオプション，211, 355–357
-xregs コンパイラオプション，355
-xrestrict, コンパイラオプション，357
-xs, コンパイラオプション，359
-xsafe=mem, コンパイラオプション，359–360
-xspace, コンパイラオプション，360
-xtarget=*t*, コンパイラオプション，360–364
-xtime, コンパイラオプション，365
-xtrigraphs, コンパイラオプション，365
-xunroll=*n*, コンパイラオプション，366–367
-xustr, コンパイラオプション，367
-xvector, コンパイラオプション，368
-xvis, コンパイラオプション，369
-xvpara, コンパイラオプション，369
-xwe, コンパイラオプション，370
X 型挿入子、iostream，178

Z

-z arg, コンパイラオプション，371

あ

アクセシブルな製品ドキュメント，20
アセンブラ、コンパイル構成要素，41
アセンブリ言語のテンプレート，369
値
  cout への挿入，178
  double，200
  float，178
  long，191
  マニピュレータ，177, 191
値クラス, 使用，113–115
アプリケーション
  マルチスレッドで使用しても安全，121
  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，134–136
  マルチスレッドプログラムのリンク，117, 124

い

依存関係, C++ 実行時ライブラリに依存しないようにする，212
インスタンス化
  オプション，99
  テンプレート関数，85
  テンプレート関数メンバー，86
  テンプレートクラス，85–86
  テンプレートクラスの静的データメンバー，86
インスタンス化の方法, テンプレート，99
インスタンスの状態, 一致した，104
インスタンスメソッド
  静的，101
  大域，102
  半明示的，103
  明示的，102
インライン関数
  C++, 使用に適した状況，112
  オプティマイザによる，310
インライン展開、アセンブリ言語テンプレート，41

え

エラー
  状態、iostream，179
  ビット，180
  マルチスレッドでの安全性のチェック，124
エラー処理
  complex，203
  入力，183–184
エラーメッセージ
  complex_error，202
  コンパイラのバージョンの非互換性，37
  リンカー，38, 40
演算子
  complex，204
  iostream，178–180, 181
  基本算術，201
  スコープ決定，126
演算ライブラリ、複素数，199–200

お

オーバーヘッド, マルチスレッドで使用しても安全なクラスのパフォーマンス，126, 127
オブジェクト
  stream_locker，133
  一時オブジェクト, 寿命，233
  一時，111
  共有オブジェクトの破棄，134
  大域共有，130
  との処理の問題に対処するもっとも簡単な方法，131
  破棄する順序，233
  ライブラリ, リンク時，207
オブジェクトスレッド, private，131
オブジェクトファイル
  er_src によるコンパイラのコメントの読み取り，293
  再配置可能，209
  リンク順序，48
オブジェクトファイル内のコンパイラのコメント、er_src による読み取り，293
オプション, コマンド行
  「アルファベット順リストの個々のオプション」を参照

オプション, コマンド行 (続き)
C++ コンパイラオプションのリファレンス, 216–371
機能別に要約, 48–61
構文, 47
オプティマイザのメモリー不足, 45

か

外部
インスタンス, 99
リンケージ, 100
拡張機能, 65–73
定義, 30
非標準コードを許可する, 232
数, 複素数, 199
数の共役, 199
仮想メモリー、制限, 44
各国語のサポート, アプリケーション開発, 31
活性文字列, 340
カバレッジ分析 (tcov), 352
ガベージコレクション
ライブラリ, 143, 149
環境変数
CCFLAGS, 45–46
LD_LIBRARY_PATH, 208
SUNWS_CACHE_NAME, 103
関数
streambuf 公開仮想関数, 133
置き換え, 68
オプティマイザによるインライン化, 310
静的、クラスフレンドとして, 71–72
宣言指定子, 65
動的 (共有) ライブラリ, 209
マルチスレッドで使用しても安全な公開関数, 123
関数、__func__ における名前, 72
関数テンプレート, 81–82
「テンプレート」も参照
使用, 82
宣言, 81
定義, 82
関数の並べ替え, 307
関数レベルの並べ替え, 307

き

キャッシュ
オプティマイザ用, 294
ディレクトリ, テンプレート, 37
境界整列
デフォルト, 382
もっとも厳密な, 381–382
共有オブジェクト, 131, 134
共有体宣言指定子, 66
共有ライブラリ
C プログラムからのアクセス, 212
構築, 209–210, 245
構築, 例外のある, 110
名前, 248
のリンクを使用できなくする, 222
例外を含む, 210
極座標, 複素数, 199
局所スコープ規則、使用する/使用しない, 232

く

空白
行頭, 182
抽出子, 183
飛ばす, 183, 190
区間演算ライブラリ, リンク, 309
クラス
渡す, 114
クラスインスタンス, 名前のない, 71
クラス宣言指定子, 66
クラステンプレート, 82–85
「テンプレート」も参照
使用, 84–85
静的データメンバー, 84
宣言, 83
定義, 83
パラメータ, デフォルト, 87
不完全な, 82
メンバー, 定義, 83–84
クラスライブラリ, 使用, 146–149

け
警告
C ヘッダーの置き換え，155
移植性がないコード，281
移植性を損なう技術的な違反，282
効率が悪いコード，281
認識できない引数，40
廃止されている構文，282
問題がある ARM 言語構造，233
抑止，282
言語
C99 サポート，317
各国語のサポート，31
言語が混合したリンク，317
検索, テンプレート定義ファイル，105
検索パス
インクルードファイル, 定義，249
定義，105
動的ライブラリ，146
標準ヘッダーの実装，153

こ
公開関数, マルチスレッドで使用しても安全，123
構成マクロ，142
構造体宣言指定子，66
構文
CC コマンド行，35
オプション，47
コードオプティマイザ, コンパイル構成要素，41
コード生成, インライン機能とアセンブラ、コンパイル構成要素，41
コードの最適化, -fast による，228
互換モード
「-compat」も参照，219
iostream，175
libC，175, 177
libcomplex，199
Tools.h++，143
国際化、実装，31
コピー
ストリームオブジェクト，187
ファイル，193
コマンド行
オプション, 認識できない，39
認識されるファイル接尾辞，35
混合算術演算, 複素数演算ライブラリ，205
コンストラクタ
complex クラス，200–201
iostream，176
静的，209
コンパイラ
構成要素の起動順序，40
診断，39–40
バージョン, 非互換性，37
コンパイル, メモリー条件，43–45
コンパイルとリンク，37

さ
サイズ, メモリー，382
最適化
-fast による，228
pragma opt，381
-xmaxopt，326
数学ライブラリ，321
対象ハードウェア，360
リンク時，322
レベル，331
先読み命令、有効化，346
サブプログラム、コンパイルオプション，38
三角関数, 複素数演算ライブラリ，203

し
シーケンス、マルチスレッドで使用しても安全な入出力操作の実行，130–131
シェル, 仮想メモリーの制限，44
$ 識別子, 最初以外の文字に許可する，232
シグナルハンドラ
およびマルチスレッド，118
例外，107
実行時エラーメッセージ，108
実行時ライブラリの README，173
実部, 複素数，199
シフト演算子、iostream，189

従来のリンクエディタ、コンパイル構成要素，41
終了関数，378
出力，175
　エラーの処理，179–180
　バイナリ，180
　バッファーのフラッシュ，180
初期化関数，379
指令 (プラグマ)，374–385
シンボル宣言指定子，65
シンボルテーブル, 実行可能ファイル，272

す

数学関数、複素数演算ライブラリ，202
数学ライブラリ, オプティマイズされた
　バージョン，321
スコープ決定演算子, unsafe_ クラス，126
スタック, のページサイズを設定する，336
ストリーム、定義，196
スペル、代替，231
スワップ領域，43

せ

制限付きポインタ，358
静的
　アーカイブライブラリ，207
　インスタンス (非推奨)，99
　オブジェクト、非ローカルの初期設定子，233
　関数, 参照，93
　データ, マルチスレッドアプリケーション
　　内，130
　変数, 参照，93
静的なリンク
　コンパイラ提供ライブラリ，145, 273
　デフォルトライブラリ，149–150
　テンプレートインスタンス，101
　ライブラリの結合，217
制約、マルチスレッドで使用しても安全な
　iostream，124–126
絶対値, 複素数，199
接尾辞
　.SUNWCCh，153
接尾辞 (続き)
　コマンド行ファイル名，35
　なしのファイル，153
　ライブラリ，207
宣言指示子, __thread，67
宣言指定子
　__global，66
　__hidden，66
　__symbolic，66

そ

相互排他領域, 定義，132
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，127
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，133
挿入
　演算子，178
　定義，196
挿入演算子
　complex，204
　iostream，178–180
ソースファイル
　位置規約，105
　リンク順序，48

た

大域
　インスタンス，99
　データ, マルチスレッドアプリケーション
　　内，130
　マルチスレッドで使用しても安全なアプリ
　　ケーション内の共有オブジェクト，130
　リンケージ，100
代入, iostream，187

ち

中間言語トランスレータ, コンパイル構成要素，41

抽出
  char*，181–182
  演算子，180–181
  空白，183
  定義，196
  ユーザー定義 iostream，181

て

定義,テンプレートの検索，105
定義済みマニピュレータ,iomanip.h，189
定義取り込みモデル，77
定義分離モデル，78
定数文字列を読み取り専用メモリーに，232
データ型、複素数，199
適用子,引数付きマニピュレータ，191
デストラクタ,静的，209
デバッガデータ形式，302
デバッグ
  オプション，50–51
  プログラムの準備，39, 247
デフォルト演算子,使用，113
デフォルトライブラリ,静的なリンク，149–150
テンプレート
  入れ子，86
  インスタンス化の方法，99
  インスタンスメソッド，104
  インライン，320
  キャッシュディレクトリ，37
  コマンド，97
  コンパイル，100
  冗長コンパイル，97
  静的オブジェクト,参照，93
  ソースファイル，105
  定義検索時の問題の回避，106
  定義分離型と定義取り込み型の編成，105
  特殊化，87–89
  標準テンプレートライブラリ (STL)，157
  部分特殊化，88–89
  リポジトリ，103
  リンク，39
テンプレート定義
  検索パス，105
  定義検索時の問題の回避，106
テンプレート定義 (続き)
  取り込み型編成，77
  分離，78
  分離された,ファイル，105
テンプレートのインスタンス化，85–86
  暗黙的，85
  関数，85–86
  全クラス，98
  明示的，85–86
テンプレートのプリリンカー、コンパイル構成要素，41
テンプレートの問題，89–95
  静的オブジェクト,参照，93
  定義検索時の問題の回避，106
  テンプレート関数のフレンド宣言，90–92
  テンプレート定義内での修飾名の使用，92
  引数としての局所型，90
  非局所型名前の解決とインスタンス化，89

と

動的 (共有) ライブラリ，150–151, 209–210, 217, 248
ドキュメント、アクセス，19–20
ドキュメント索引，19
トラップモード，244

な

内部手続きオプティマイザ，313
名前のないクラスインスタンス,受け渡し，71

に

入出力、複雑，175
入力
  iostream，180–181
  エラー処理，183–184
  先読み，183
  バイナリ，182
入力/出力,complex，204
入力データの先読み，183

は

廃止されている構文、許可しない，231
配置、テンプレートのインスタンス，99
バイナリ最適化，292
バイナリ入力, 読み込み，182
バッファー
　出力のフラッシュ，180
　定義，196
パフォーマンス
　-fast による最適化，228
　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，126, 127
半明示的インスタンス，99, 102–103

ひ

ヒープ、のページサイズを設定する，336
引数付きのマニピュレータ、iostream，190–191
引数なしのマニピュレータ、iostream，189–190
非互換性, コンパイラのバージョン，37
左シフト演算子
　complex，204
　iostream，178
非標準機能
　定義，30
　非標準コードを許可する，232
非標準の機能，65–73
『標準 C++ ライブラリユーザーズガイド』，158
標準 iostream クラス，175
標準エラー, iostream，175
標準出力, iostream，175
標準、準拠，30
標準ストリーム, iostream.h，130
標準テンプレートライブラリ (STL)，157
標準入力, iostream，175
標準ヘッダー
　置き換え，154–155
　実装，153–155
標準モード
　「-compat」も参照，219
　iostream，175, 177
　libCstd，199
　Tools.h++，143

ふ

ファイル
　C の標準ヘッダーファイル，153
　fstream の使用，184–187
　位置の再設定，186–187
　オープンとクローズ，186
　オブジェクト，37, 48, 209
　コピー，185, 193
　実行可能プログラム，37
　標準ライブラリ，153
　複数のソース, 使用，36
ファイル記述子、使用，186
ファイル内の位置の再設定、fstream，186–187
ファイル名
　.SUNWCCh ファイル名接尾辞，153
　接尾辞，35
　テンプレート定義ファイル，105
フォーマットの制御, iostream，187
複数のソースファイル, 使用，36
複素数, データ型，199
浮動小数点
　精度 (Intel)，243
　不正，244
プラグマ (指令)，374–385
プリコンパイル済みヘッダーファイル，339
プリプロセッサ, に対してマクロを定義する，222
プログラム
　基本的構築手順，33–34
　マルチスレッドプログラムの構築，117–118
プロセッサ、対象システムを指定する，360
プロファイル, -xprofile，350
フロントエンド, コンパイル構成要素，41

へ

並列化
　警告メッセージの有効化，369
　複数プロセッサのための -xautopar を使用した
　　並列化，292
並列化 -xreduction，354
ページサイズ、スタックとヒープ用に設定す
　る，336
べき等，75

ヘッダーファイル
complex，206
C 標準，153
iostream，130, 177, 189
言語に対応した，75–76
作成，75–77
標準ライブラリ，151, 158–159
べき等，76–77
別名, によるコマンドの簡略化，45
偏角, 複素数，199
変更可能な引数のリスト，41–42
変数, スレッド固有の記憶領域指示子，67
変数宣言指定子，65
変数のスレッド固有の記憶領域，67

ま
マニピュレータ
iostream，188–191
事前定義，188–189
引数なし，189–190
マニュアルページ
C++ 標準ライブラリ，159–172
complex，206
iostream，175, 185, 187, 191
アクセス，31, 144
マルチスレッド
アプリケーション，117
コンパイル，117–118
例外処理，118
マルチスレッド化，264–265
マルチスレッドで使用しても安全
アプリケーション，121
オブジェクト，121
クラス, 派生における注意事項，133
公開関数，123
パフォーマンス, オーバーヘッド，127
パフォーマンスオーバーヘッド，126
ライブラリ，121
マルチメディアタイプ、の処理，369

み
右シフト演算子
complex，204
iostream，180

め
メイクファイルの依存関係，325
明示的インスタンス，99
メモリーサイズ，382
メモリー条件，43–45
メモリーバリアー組み込み関数，136–137
メンバー変数, キャッシュ，115–116

も
文字、単一読み込み，182

ゆ
ユーザー定義型, マルチスレッドで使用しても安全，125
ユーザー定義の型, iostream，178
優先順位、の問題回避，178

ら
ライブラリ
C++ コンパイラ, に添付，141
C++ 標準，157
C インタフェース，141
-mt によるリンク，141
Sun Performance Library、リンク，258, 321
置き換え, C++ 標準ライブラリの置き換え，151–155
共有，150–151, 222
共有ライブラリの構築，301
共有ライブラリの名前，248
区間演算，309
クラス, 使用，146
構成マクロ，142

ライブラリ (続き)
最適化された数学ルーチン，321
従来型の iostream，175
使用，141–155
接尾辞，207
とは，207–208
リンクオプション，148–149
リンク順序，48
ライブラリ, 構築
C API を持つ，211–212
公開，211
静的 (アーカイブ)，207–212
動的 (共有)，207
非公開，210
リンクオプション，246
例外を含む共有，210

り

リテラル文字列を読み取り専用メモリーに，232
リンカースコープ，65
リンク
complex ライブラリ，147–148
-mt オプション，124
コンパイルからの分離，38
コンパイルとの整合性，38–39
システムライブラリを使用しない，329
シンボリック，153
静的 (アーカイブ) ライブラリ，145, 149–150, 207, 217, 273
テンプレートのインスタンス化の方法，99
動的 (共有) ライブラリ，208, 217
マルチスレッドで使用しても安全な libC ライブラリ，124
ライブラリ，141, 144, 148–149
リンク時の最適化，322

る

ループ，303
-xloopinfo，323
縮約 -xreduction，354

れ

例外
longjmp および，109–110
setjmp および，109–110
およびマルチスレッド，118–119
関数, 置き換えでの，68
共有ライブラリ，210
許可しない，232
シグナルハンドラおよび，109–110
事前定義，108–109
トラップ，244
のある共有ライブラリの構築，110
標準クラス，109
標準ヘッダー，108
無効化，108

ろ

ロック
「stream_locker」も参照
streambuf，123
オブジェクト，131–133
相互排他，127, 133

わ

ワークステーション, メモリー要件，45